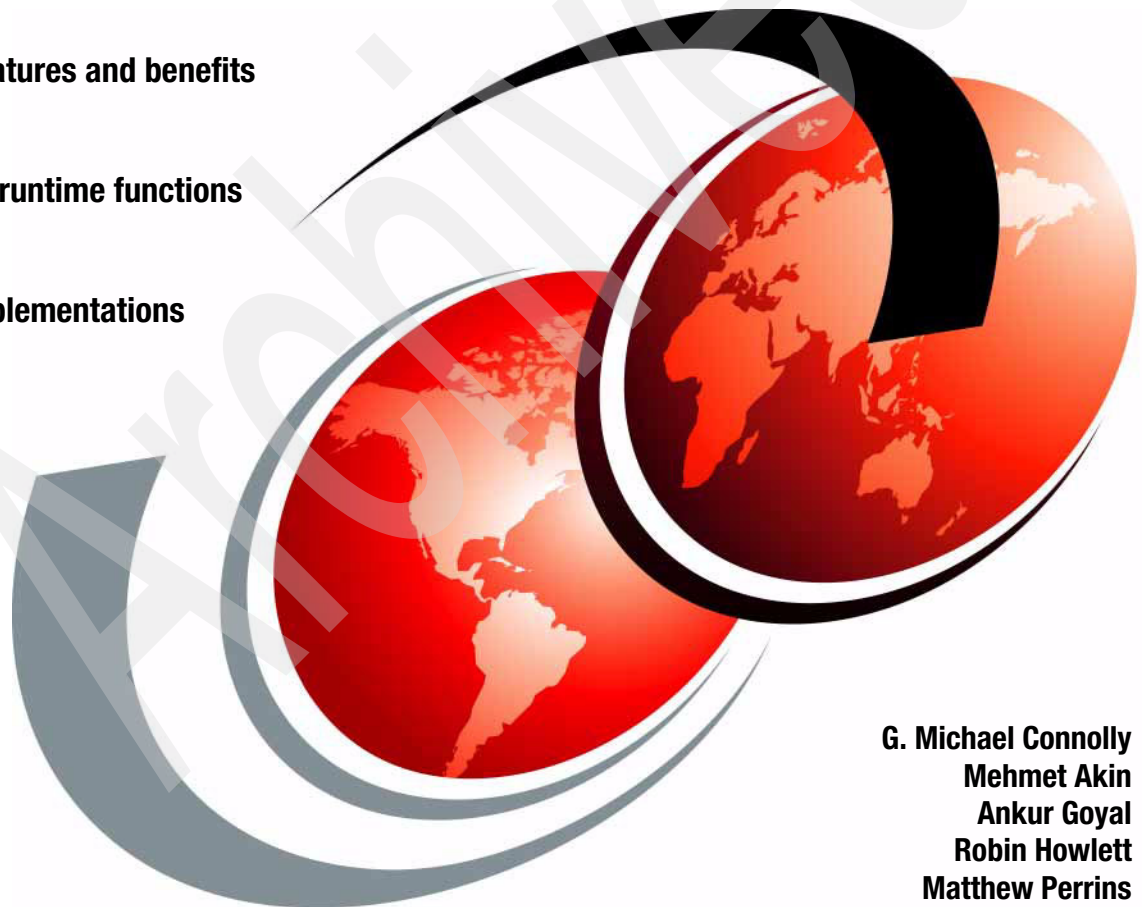


# Building Dynamic Ajax Applications Using WebSphere Feature Pack for Web 2.0

Web 2.0 features and benefits explained

Ajax client runtime functions explored

Sample implementations presented



G. Michael Connolly  
Mehmet Akin  
Ankur Goyal  
Robin Howlett  
Matthew Perrins





International Technical Support Organization

**Building Dynamic Ajax Applications Using  
WebSphere Feature Pack for Web 2.0**

November 2008

Archived

**Note:** Before using this information and the product it supports, read the information in “Notices” on page xiii.

### **First Edition (November 2008)**

This edition applies to WebSphere Application Server Feature Pack for Web 2.0 for Version 6.0.2, Version 6.1, and Community Edition Version 2.0.

© Copyright International Business Machines Corporation 2008. All rights reserved.

Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

# Contents

<b>Notices</b> .....	xiii
Trademarks .....	xiv
<b>Preface</b> .....	xv
The team that wrote this book .....	xvi
Become a published author .....	xviii
Comments welcome .....	xviii
<b>Chapter 1. Introduction to Web 2.0</b> .....	1
1.1 Introduction to Web 2.0 .....	2
1.2 Catching a wave of innovation .....	2
1.2.1 A tipping point .....	2
1.2.2 Web 2.0 technologies .....	3
1.3 Web 2.0 releases the value of SOA .....	4
1.4 Enterprise Web 2.0 .....	8
1.5 Extending the reach of SOA with RESTful SOA .....	9
1.5.1 REST and the foundation of the RESTful SOA .....	10
1.5.2 Unleash your assets .....	15
1.5.3 Unleashing your J2EE assets .....	16
1.5.4 Classification of feeds .....	17
1.6 What is Ajax .....	18
1.6.1 History .....	18
1.6.2 Justification .....	19
<b>Chapter 2. Feature pack overview</b> .....	21
2.1 Feature packs for WebSphere Application Server .....	22
2.2 Feature Pack for Web 2.0 features .....	24
2.2.1 Ajax connectivity .....	24
2.2.2 Server-side libraries .....	25
2.2.3 Ajax client runtime (Ajax Development Toolkit) .....	26
2.2.4 Web 2.0 sample applications .....	28
2.2.5 FeedSphere sample .....	29
2.3 Benefits of Feature Pack for Web 2.0 .....	29
2.4 References .....	29
<b>Chapter 3. Core concepts</b> .....	31
3.1 The technical foundations of Web 2.0 .....	32
3.1.1 HTTP Protocol .....	33
3.1.2 REST .....	34

3.1.3 Asynchronous JavaScript (Ajax)	35
3.1.4 JSON	35
3.1.5 POX (XML)	36
3.1.6 Atom	36
3.1.7 JavaScript	36
3.1.8 DOM	37
3.1.9 DHTML	37
3.1.10 CSS	38
3.2 Model View Controller	38
3.2.1 Model 2 J2EE MVC	39
3.2.2 Web 2.0	40
<b>Chapter 4. Installation</b>	<b>43</b>
4.1 Supported platforms	44
4.2 Supported Integrated Development Environments (IDEs)	44
4.3 Installing the application server product	44
4.3.1 Updating the application server	45
4.3.2 Applying the appropriate iFix for the Web messaging feature	45
4.4 Downloading Feature Pack for Web 2.0	45
4.4.1 WebSphere Application Server Community Edition Version 2.0	46
4.4.2 WebSphere Application Server Version 6 for Distributed, i5/OS	46
4.4.3 WebSphere Application Server Version 6 for z/OS	46
4.5 Installing Feature Pack for Web 2.0	46
4.5.1 System requirements	46
4.5.2 Installing Feature Pack for Web 2.0 on distributed OSs	47
4.5.3 Installing Feature Pack for Web 2.0 on i5/OS	50
4.5.4 Installing Feature Pack for Web 2.0 on z/OS	53
4.6 Uninstalling Feature Pack for Web 2.0	53
4.6.1 Uninstalling Feature Pack for Web 2.0	53
4.6.2 Uninstalling the feature pack on i5/OS	56
4.6.3 Manual uninstallation	57
4.6.4 Troubleshooting	60
<b>Chapter 5. Application development tools</b>	<b>63</b>
5.1 Ajax development tools overview	64
5.2 Rational Application Developer 7.0.0.6	64
5.2.1 IBM Installation Manager	64
5.2.2 Installation of Rational Application Developer	66
5.2.3 Upgrading IBM Installation Manager	72
5.2.4 Upgrading IBM Rational Application Developer	73
5.2.5 Configuring Web 2.0 feature pack Eclipse plug-in	78
5.2.6 Uninstalling IBM Rational Application Developer	81

5.3 Eclipse 3.3 . . . . .	81
5.3.1 Eclipse installation . . . . .	82
5.3.2 Configuration of Eclipse for WAS CE server . . . . .	89
5.4 Firebug 1.05 . . . . .	92
5.4.1 Installation . . . . .	92
5.4.2 IE debugging aids . . . . .	93
5.5 JSON Formatter . . . . .	93
<b>Chapter 6. Ajax connectivity . . . . .</b>	<b>97</b>
6.1 Web 2.0 to SOA connectivity overview . . . . .	98
6.2 RPC Adapter . . . . .	99
6.2.1 Protocols supported by RPC Adapter . . . . .	100
6.2.2 Sample RPC Adapter implementation . . . . .	100
6.2.3 RPC Adapter library features . . . . .	114
6.2.4 Accessing EJBs through RPC Adapter . . . . .	126
6.2.5 XML schema for RPCAdapterConfig.xml . . . . .	126
6.2.6 Limitations of the RPC Adapter libraries . . . . .	126
6.2.7 The CourierApp sample . . . . .	126
6.2.8 Overview of the CourierApp sample . . . . .	127
6.2.9 CourierApp sample prerequisites . . . . .	127
6.2.10 Limitations of the CourierApp sample . . . . .	127
6.2.11 Installing the CourierApp sample application . . . . .	128
6.3 Ajax Proxy . . . . .	131
6.3.1 Why we need it . . . . .	131
6.3.2 How it is implemented . . . . .	132
6.3.3 Ajax Proxy architecture and configuration . . . . .	134
6.3.4 Ajax proxy sample standalone implementation . . . . .	136
6.3.5 XML Schema of profile-config.xml . . . . .	141
6.3.6 Ajax Proxy limitations . . . . .	141
<b>Chapter 7. Java libraries . . . . .</b>	<b>143</b>
7.1 Java libraries . . . . .	144
7.2 Overview . . . . .	144
7.3 JSON4J . . . . .	145
7.3.1 Introduction . . . . .	145
7.3.2 JSON format . . . . .	147
7.3.3 Consuming JSON in a browser . . . . .	150
7.3.4 Create a project . . . . .	151
7.3.5 Web project configuration . . . . .	155
7.3.6 Configuring WebSphere shared libraries . . . . .	157
7.3.7 Sample invocation servlet . . . . .	158
7.3.8 JSON object and JSON array . . . . .	162
7.3.9 Using JSON within a browser . . . . .	164

7.3.10	JSON arrays	167
7.3.11	Java objects to JSON	169
7.3.12	JSON4J limitations	175
7.4	Feeds libraries (Apache Abdera)	175
7.4.1	Package structure	176
7.4.2	Atom 1.0 Syndication Format	178
7.4.3	Atom Publishing Protocol (APP)	182
7.4.4	Apache Abdera	183
7.4.5	FeedSphere sample	185
7.4.6	Atom Publishing Protocol and HTTP operations	193
7.4.7	AtomPub server implementation using Abdera	203
7.4.8	Atom Publishing Protocol support	205
7.4.9	Abdera APP client	225
<b>Chapter 8.</b>	<b>REST</b>	<b>231</b>
8.1	Introduction to REST	232
8.2	Building a REST servlet	235
8.3	REST-as-CRUD or REST-as-protocol	260
8.4	REST or SOA: Making the decision	261
8.4.1	Reliable messaging	263
8.4.2	Transactional REST	264
8.5	REST and Ajax	265
8.6	REST in the industry	267
8.6.1	JAX-RS (JSR 311)	267
8.6.2	Jersey	267
8.6.3	Restlet	267
<b>Chapter 9.</b>	<b>Ajax client runtime</b>	<b>269</b>
9.1	Introduction to the Dojo Toolkit	270
9.1.1	Overview of the Ajax client runtime	270
9.1.2	Overview of the Dojo Toolkit	271
9.1.3	Dojo Foundation, licensing, and help	273
9.2	Ajax client runtime directory structure	274
9.2.1	WebSphere 6.x servers	275
9.2.2	WebSphere Server Community Edition and z/OS	276
9.3	Adding the Ajax client runtime to your project	276
9.3.1	Importing the libraries into the project folder	277
9.3.2	Including the Dojo library	278
9.4	The Dojo core library	278
9.4.1	The dojo folder structure	278
9.4.2	Dojo's packaging system	279
9.4.3	Dojo configuration with djConfig	282
9.4.4	Useful Dojo core functions	285



9.4.5	Object-oriented programming with Dojo	292
9.4.6	Visual effects	296
9.4.7	Cookie handling	301
9.4.8	Dojo's event system	302
9.4.9	Drag and drop	307
9.4.10	Handling the back-button problem	308
9.4.11	Selecting DOM nodes with Dojo's query engine	312
9.5	Dojo's pub/sub event-system	313
9.6	XHR, IFrames, JSONP, and RPC	316
9.6.1	XMLHttpRequest	316
9.6.2	JSONP cross-domain scripting with Dojo	323
9.6.3	Uploading files with IFrames	325
9.6.4	Remote Procedure Calls	327
9.7	The Dojo widget library	328
9.7.1	Widget basics	329
9.7.2	Overview of existing widgets	329
9.7.3	Using widgets declaratively	337
9.7.4	Widget HTML attributes	340
9.7.5	Extension points and declarative event connections	340
9.7.6	Instantiating widgets programmatically	341
9.7.7	Custom widget styling	343
9.7.8	Overriding Dojo styles	347
9.7.9	Finding the correct class to override	349
9.7.10	Accessibility and Internationalization through CSS	350
9.8	Creating custom Dojo widgets	350
9.8.1	Setting up the folders	350
9.8.2	Important functions during the widget life cycle	352
9.8.3	Widget basics	353
9.8.4	Declaring a widget class declaratively	359
9.8.5	Using the Dojo parser	361
9.8.6	Extending widgets	363
9.8.7	Advanced widgets	366
9.9	Data API	371
9.9.1	Core design concepts	371
9.9.2	Datastores and dojo.data APIs	372
9.9.3	Read API	377
9.9.4	Write API	378
9.9.5	Identity API	380
9.9.6	Notification API	381
9.9.7	Query	383
9.9.8	Request API	384
9.9.9	Data formats	384

9.10 Dojo extensions in dojoX . . . . .	388
9.11 Unit testing with the Dojo Objective Harness . . . . .	390
9.11.1 Writing tests . . . . .	390
9.11.2 Running tests . . . . .	397
9.12 IBM Extensions to the Dojo Toolkit . . . . .	398
9.12.1 IBM Atom library . . . . .	398
9.12.2 IBM Gauge widgets . . . . .	410
9.12.3 IBM OpenSearch library . . . . .	416
9.12.4 IBM SOAP library . . . . .	420
9.13 Adding the Web 2.0 FEP to existing applications . . . . .	423
9.13.1 Browser-side enhancement . . . . .	426
9.13.2 Server-side enhancement . . . . .	434
9.14 Sample: PlantsByWebSphere application . . . . .	438
9.14.1 Installation and configuration . . . . .	438
9.14.2 Usage . . . . .	448
<b>Chapter 10. Web messaging service . . . . .</b>	<b>455</b>
10.1 Introduction . . . . .	456
10.2 Evolution of event-driven Web applications . . . . .	456
10.2.1 Web applications circa 1997 . . . . .	456
10.2.2 Web applications circa 2005 . . . . .	458
10.3 Comet . . . . .	461
10.4 Bayeux Protocol . . . . .	463
10.5 Cometd and the Dojo Toolkit . . . . .	465
10.6 Web messaging service overview . . . . .	466
10.6.1 Scaling using channels . . . . .	468
10.6.2 Service integration bus connectivity . . . . .	469
10.7 Web messaging service challenges . . . . .	471
10.7.1 Web servers . . . . .	471
10.7.2 Two-connection limit . . . . .	472
10.8 Using the Web messaging service . . . . .	472
10.9 Web messaging service Quick Start . . . . .	473
10.10 Enabling the Web messaging service . . . . .	478
10.11 Developing Web messaging enabled applications . . . . .	480
10.11.1 Web messaging utility library . . . . .	480
10.11.2 Web messaging servlet . . . . .	480
10.12 Web messaging service configuration . . . . .	482
10.12.1 Service integration bus configuration for Web messaging service	483
10.12.2 Bayeux configuration . . . . .	490
10.12.3 Web messaging service client reference . . . . .	492
10.13 Publishing messages to the Web messaging service . . . . .	493
10.13.1 Browser client publishing . . . . .	493
10.13.2 Publishing through a Web application . . . . .	493

10.14 Clustered Web messaging enabled applications . . . . .	495
10.14.1 Application install . . . . .	495
10.14.2 Workload management . . . . .	495
10.15 Securing Web messaging enabled applications . . . . .	496
10.15.1 Protecting a Web messaging URI . . . . .	496
10.15.2 Service integration bus authentication and authorization . . . . .	497
10.16 Service integration. . . . .	499
10.16.1 Bayeux channel-to-service integration bus channel mapping . . . . .	499
10.16.2 JMS object types support . . . . .	499
10.17 Logging and trace for the Web messaging service . . . . .	500
10.18 Troubleshooting the Web messaging service . . . . .	500
10.18.1 Runtime enablement reference . . . . .	500
10.18.2 Client error message reference . . . . .	501
10.19 Sample: QuoteStreamer sample application . . . . .	503
10.19.1 Pre-installation configuration . . . . .	504
10.19.2 Installation through the administrative console . . . . .	517
10.19.3 Installation through Rational Application Developer . . . . .	519
10.19.4 Accessing to the installed application . . . . .	523
10.19.5 How the QuoteStreamer sample application works . . . . .	524
<b>Chapter 11. Security . . . . .</b>	<b>539</b>
11.1 Overview . . . . .	540
11.2 Attack scenarios and the effect . . . . .	540
11.2.1 Cross-site scripting (XSS) . . . . .	540
11.2.2 Cross-site Request Forgery (CSRF) . . . . .	541
11.3 Recommended best practices . . . . .	542
11.3.1 Server side . . . . .	542
11.3.2 Browser side . . . . .	542
11.4 Securing RPCAdapter (application server security) . . . . .	544
11.5 Tools . . . . .	550
11.5.1 Rational Application Developer . . . . .	550
11.5.2 WebSphere DataPower XML Security Gateway XS40 . . . . .	551
11.5.3 Rational AppScan . . . . .	553
<b>Chapter 12. Best practices . . . . .</b>	<b>555</b>
12.1 Introduction . . . . .	556
12.2 ShrinkSafe and the Dojo Build System . . . . .	556
12.2.1 Why ShrinkSafe . . . . .	556
12.2.2 Compressing a single file . . . . .	558
12.2.3 Dojo build and layering system . . . . .	565
12.2.4 Creating a custom build . . . . .	566

12.3	JavaServer Pages for JSON	577
12.4	JSON compression	579
12.5	Internationalization (i18n)	579
12.5.1	Dojo i18n	580
12.5.2	Using the JSP Standard Tag Library (JSTL)	584
12.5.3	Formatting strings with dojo.string.substitute	588
12.5.4	I18n support for strings in Dojo widgets	588
12.5.5	Adding additional language support to Dijit	588
12.5.6	Currency, number, and date formatting	590
12.5.7	String encoding	592
12.5.8	UTF-8 encoding for the client side	592
12.5.9	UTF-8 encoding on the server side	593
12.5.10	Dojo and UTF-8 encoding of request parameters	593
12.5.11	Bidirectional (BiDi) layout of elements	594
12.6	Accessibility	599
12.6.1	Device-independent interaction	599
12.6.2	Sizing issues	600
12.6.3	High-contrast mode and turning off images	600
12.6.4	Assistive technology support	602
12.6.5	Accessibility support in available Dojo widgets	604
<b>Chapter 13</b>	<b>Rich Internet applications</b>	<b>605</b>
13.1	Rich Internet applications	606
13.1.1	Benefits of an RIA	607
13.1.2	Limitations	608
13.1.3	RIA run times	610
13.2	Integrating RIAs with WebSphere Application Server	613
13.3	Mobile devices	613
13.3.1	Lotus Expeditor for devices	614
13.3.2	iPhone SDK	615
13.3.3	Android SDK	616
13.4	Adobe AIR	617
13.4.1	Scenario	617
13.4.2	Server	619
13.4.3	Client	627
13.5	Summary	642
<b>Appendix A</b>	<b>XML schema definitions</b>	<b>645</b>
<b>Appendix B</b>	<b>Additional material</b>	<b>653</b>
	Locating the Web material	653
	Using the Web material	654
	How to use the Web material	654

**Related publications** ..... 655

IBM Redbooks publications ..... 655

Other publications ..... 655

Online resources ..... 656

How to get Redbooks publications ..... 658

Help from IBM ..... 658

Archived



# Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information about the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

*IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.*

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.


## COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

# Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at <http://www.ibm.com/legal/copytrade.shtml>

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

AIX®	IBM®	Redbooks®
CICS®	iSeries®	Redbooks (logo)  ®
Cross-Site®	Lotus®	Sametime®
DataPower®	Passport Advantage®	Tivoli®
DB2®	pSeries®	WebSphere®
developerWorks®	Quickr™	z/OS®
i5/OS®	Rational®	zSeries®

The following terms are trademarks of other companies:

Adobe Flash, Adobe Flex, Adobe, Dreamweaver, Flex Builder, MXML, and Portable Document Format (PDF) are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, other countries, or both.

EJB, Enterprise JavaBeans, J2EE, J2SE, Java, JavaBeans, JavaFX, JavaScript, JavaServer, JDBC, JDK, JNI, JSP, JVM, Solaris, Sun, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Internet Explorer, JScript, Microsoft, Outlook, PowerPoint, Silverlight, Windows Mobile, Windows Vista, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.



# Preface

This IBM® Redbooks® publication positions the capabilities of the new Web 2.0 Feature Pack for the WebSphere® Application Server. It is very much directed at Web 2.0 developers who want information that enables them to build a dynamic Ajax application for the WebSphere platform. This book covers all the major topics that the Web 2.0 FEP supports and gives clear and detailed examples of how to enable these capabilities within your own applications.

## **Ajax runtime**

The Ajax runtime is built upon the very successful Dojo Toolkit and offers a powerful toolkit for building rich Internet applications. It contains a range of extra features that extend the powerful framework and, combined with WebSphere support, offers the best-of-breed platform for your next application.

## **Feeds**

Delivering dynamic data into mashup will be the norm very soon and using the rich ATOM support you can quickly and easily produce dynamic feeds that can be combined in your mashup solutions very quickly.

## **Messaging**

The Web 2.0 FEP supports COMET, which enables a very powerful publish and subscribe framework for next-generation live-feed applications. Everything from stock quotes to sports scores could be delivered dynamically to your user's desktop using the message capabilities.

## **Ajax proxy and remote procedure call (RPC) adapter**

The Ajax proxy and RPC Adapter offer flexible extensions to the already powerful servlet support within the Web container. They allow representational state transfer (REST)-based applications to invoke services local to the server or be redirected to remote services through the proxy. REST support will become a norm in the next generation of Web applications.

This book provides clear insight into how to design, install, and deploy applications that take advantage of these modern Web 2.0 programming techniques. The feature pack extends the market-leading WebSphere application server with all the capabilities that it needs to enable the most powerful Web 2.0 experience possible.

Even if you just install the samples, look at what the Web 2.0 FEP can offer your organization today.

## The team that wrote this book

This book was produced by a team of specialists from around the world working at the International Technical Support Organization, Poughkeepsie Center.

**G. Michael Connolly** is an IT consultant at the ITSO, Poughkeepsie Center. He has more than 30 years of IBM software development experience in both distributed systems and the mainframe zSeries®. He holds a BA in humanities from Villanova University. His areas of expertise include TCP/IP communications, UNIX® System Services, EWLM®, and WebSphere for z/OS®.

**Mehmet Akin** is a Software Engineer in the Lotus® Sametime® Core Services team based in the Dublin Software Lab (Ireland). He obtained his degree in Computer Science at the University of Applied Sciences Fulda in Germany as the best graduate and moved to Dublin afterwards in 2007. His areas of expertise are the Dojo Toolkit, Java™ EE, mobile programming, and Web 2.0 technologies. He is fluent in English, German, and Turkish.

**Ankur Goyal** is an IT Specialist with IBM with the Rational® Software Technical Sales team in India. He consults and supports customer solutions on IBM Middleware, open source, open standards, and emerging technologies. He holds a bachelor's degree in information technology from National Institute of Technology, Durgapur, and is also IBM certified for J2EE™, DB2®, WebSphere, Rational, XML, and SOA. His areas of interest include software development best practices, Eclipse, and integrating Web technologies with the help of open standards. Prior to this role he was part of the IBM Academic Initiative-Ecosystem group, where his team was responsible for building an ecosystem around IBM Middleware in the emerging market of India. He joined IBM in 2004 after gaining expertise in application development as a Software Engineer.

**Robin Howlett** is a Software Engineer working in the Dublin Software Lab (Ireland) with the Lotus Sametime Core Services team. He holds a master's degree in computer science (networks and distributed systems) from the University of Dublin, Trinity College, and is a graduate of the Dublin Institute of Technology (2004). He has worked for IBM since 2007, and also has experience in the financial services and stock trading industry. He has particular interest and expertise in lightweight Web 2.0 technologies, distributed computing, enterprise messaging architecture, data visualization, social networks, mobile development, and open standards.

**Matthew Perrins** is an Executive IT Specialist working for the IBM Hursley Lab Services organization. He is driving the implementation of successful solutions across Europe using Web 2.0 and client technologies. He has worked for IBM since 1989 and has spent the last 15 years working with Java and enterprise solutions with WebSphere and J2EE. He leads the consumer IT practice for the Industry Solutions Lab Services team and is focused on designing and developing solutions that operate between the user and the edge of an SOA. He focuses particularly on Web 2.0, Ajax, and the rich client technologies that operate on the desktop and device-like Eclipse RCP and Lotus Expeditor, Notes, and Sametime. He focuses on the user interface technologies that are strategic to IBM and our customers.



*The Redbooks publication team, from left to right: Ankur Goyal, Matthew Perrins, G. Michael Connolly, Mehmet Akin, Robin Howlett*

Thanks to the following people for their contributions to this project:

Rick Robinson, Martin Gale, Dave Artus  
IBM UK

Kyle Brown  
IBM US

Kevin Haverlock  
IBM US

Chris Mitchell  
IBM US

Jared Jurkiewicz  
IBM US

Roland Barcia  
IBM US

Rich Conway and Peter Bertolozzi  
IBM International Technical Support Organization, Poughkeepsie Center

## Become a published author

Join us for a two- to six-week residency program! Help write a book dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You will have the opportunity to team with IBM technical professionals, Business Partners, and Clients.

Your efforts will help increase product acceptance and customer satisfaction. As a bonus, you will develop a network of contacts in IBM development labs, and increase your productivity and marketability.

Find out more about the residency program, browse the residency index, and apply online at:

[ibm.com/redbooks/residencies.html](http://ibm.com/redbooks/residencies.html)

## Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks publication form found at:  
[ibm.com/redbooks](http://ibm.com/redbooks)
- ▶ Send your comments in an e-mail to:  
[redbooks@us.ibm.com](mailto:redbooks@us.ibm.com)
- ▶ Mail your comments to:

IBM Corporation, International Technical Support Organization  
Dept. HYTD Mail Station P099  
2455 South Road  
Poughkeepsie, NY 12601-5400

# Introduction to Web 2.0

This chapter introduces the business concepts and technical elements that constitute the term *Web 2.0*.

Web 2.0 is not a single product or a single piece of technology. It is a combination of behaviors, patterns, and uses of existing technologies available for the Internet that allow for the creation of innovative solutions that bring communities and networks of people together.

This chapter covers the following topics:

- ▶ Introduction to Web 2.0
- ▶ Catching a wave of innovation
- ▶ Web 2.0 releases the value of SOA
- ▶ Enterprise Web 2.0
- ▶ Extending the reach of SOA with RESTful SOA
- ▶ What is Ajax

## 1.1 Introduction to Web 2.0

Web 2.0 is at the center of a wave of excitement concerning how enterprises—commercial or public organizations—are trying to exploit the current generation of Internet technologies. This book examines how the WebSphere Application Server products can be exploited to leverage many of the technical elements that form a modern Web 2.0 solution.

This chapter introduces the key concepts of Web 2.0 and looks at the business and technical drivers behind Web 2.0, the challenges and opportunities Web 2.0 presents to enterprises, and the relationship between Web 2.0 and service-oriented architecture (SOA).

## 1.2 Catching a wave of innovation

Web 2.0 represents a new wave in business innovation, exploiting the maturation of the Internet as a new medium for communication and commerce. While Web 2.0 is not a new trend, having existed since at least 2003, its adoption by business is in a relatively early stage, and its overall impact is still growing.

Two general interpretations of what comprises Web 2.0 are widely accepted, both of them attributed to Tim O'Reilly. The more detailed of these interpretations analyzes Web 2.0 as a set of patterns through which technology is currently being used to create and support business models using Internet technologies. The more general second interpretation provides a useful insight into the real-world application of Web 2.0 technologies. Simply put, we are seeing renewed interest in the use of maturing Internet technologies to create new ways to communicate and do business.

### 1.2.1 A tipping point

The current interest in Web 2.0 technology has arisen due to a culmination of economic and technology trends. Taken together, these trends make the social and technical characteristics of the current generation of Internet technologies a fertile source of business innovation. The trends include:

- ▶ An increasing focus throughout organizations, and not just at a leadership level, on innovative ways to improve performance as it becomes increasingly hard to achieve improvements in organizational performance through cost control or mergers and acquisitions.
- ▶ The continuous global explosion of reach and participation in collaborative, pervasive, richly featured communication technologies. Over 1 billion people

are now online, nearly 300 million through a broadband connection. Studies show that a striking number of them contribute content and interact with each other socially and economically in expanding networks rather than using the Web simply to shop or access information.

- ▶ Generational changes in attitudes to technology and the media of choice for communication, consumption, and work. The Internet is not new to younger generations who have grown up with it and are comfortable using it for social interaction, commerce, and work.
- ▶ Increasing capabilities and falling costs for the manufacture of personalized or customized products. The era of mass production is evolving into the era of mass personalization. Anything from clothes to devices to automobiles can be customized or personalized at or about its normal price point, rather than requiring expensive bespoke design or production at a premium.
- ▶ A continuous process of business optimization resulting in transformation and outsourcing. Ongoing competition in the global marketplace through successive economic cycles has forced organizations to focus continuously on business optimization. This includes the integration of the computing systems that support business operations and their increasing exposure within and beyond the enterprise boundary through SOA.

### 1.2.2 Web 2.0 technologies

In this section we discuss Web 2.0 technologies.

#### **Promoting participation: Blogs, wikis, and social computing**

Our use of Internet technology has matured, resulting in the evolution of a new communications medium in which people of any generation and from any culture can participate. For instance, personal profiles, blogs, and wikis provide simple tools that allow people to communicate without understanding underlying Web and browser technologies. Using them, people can share their interests and expertise, and create collaborative content.

By using social bookmarking, people share their links to content and services, making it easier for others to find them. By tagging and rating these links, the content of the Web is categorized and rated according to the interests of consumers. All of this has resulted in a richer, better-connected Internet in which individuals can meet and communicate with each other on topics of common interest more easily than through any previous medium. IBM Lotus Connections and IBM Lotus Quickr™ provide collaboration software incorporating these features.

## **Promoting syndication and reuse: Services, feeds, and widgets**

Supporting the applications described in the previous section is a set of technologies that have emerged over the last decade or so. Syndicated news feeds—simple streams of stories and information formatted in XML according to the RSS or Atom protocols—are now widespread and enable aggregator sites to carry a vast array of content and let individuals create their own aggregations.

Feeds are an example of services created according to RESTful principles, a set of design and implementation prescriptions that aim to result in the creation of services that are as scalable and easy to use as the Internet as a whole. Many Web sites make their content and functionality available as RESTful services so that they can be incorporated into other Web applications. At the same time, open standards and SOA advancements have made many services and information sources available through robust, secure Web services.

Finally, technologies such as Asynchronous JavaScript™ + XML (Ajax) provide a more richly featured, responsive user interfaces in a browser. IBM WebSphere Application Server Feature Pack for Web 2.0 and IBM WebSphere MQ Bridge for HTTP both provide features to enable the creation of REST services, feeds, and Ajax user interfaces. The Project Zero community development project (WebSphere sMash) is also exploring the capabilities of a lightweight application environment to deliver Web 2.0 solutions using an approach based on scripting languages.

## **Promoting agility: Situational applications and mashups**

Mashup applications put great application construction power in the hands of users to combine function and content from many sources into new applications at low costs. Frequently, such applications mix operational data with news and events, financial information, or weather. People use mashups to combine external services and information with their own data, such as their holiday plans or the customer with whom they are dealing, in an application that brings together all the information and function that they need in one place.

Social computing allows people to share mashups—or the individual feeds and widgets from which they are constructed—with others, bringing this power quickly to the hands of large audiences.

## **1.3 Web 2.0 releases the value of SOA**

From their earliest days, Web services and SOA concepts have promised a new world of composite applications, simply wired together from existing services



presented through the Internet. The progress made in applying these technologies and architectures since their inceptions is now bearing fruit with the emergence of Web 2.0.

While not always considered part of the mainstream SOA movement, RESTful services and feeds based on protocols like RSS and Atom have always served as good examples of well-defined SOA services outside the world of Web services. Their widespread availability, in addition to Web services, makes syndication and situational applications possible. So while Web 2.0 is a broad theme, it very much includes the concept of composite applications created by combining services, feeds, and widgets enabled through SOA. Correspondingly, the concepts and patterns of Web 2.0 can be used to unlock new value by organizations that have invested in SOA.

## **A new world for business**

The culmination of technology and business trends drives a number of consequences:

- ▶ The distribution of all forms of content that can be digitized is undergoing revolutionary change.

The cost of distributing content to a potentially vast global audience has collapsed as content (for example, the written word, film and video, music, numerical data, and pictures) is digitized. As a result, the number of providers of digital content is exploding, and the proportion of content that is accessed through regular push channels (print and broadcast, for example) is falling rapidly as consumers access exactly what they want, when they want it, online.

- ▶ The influence on consumers of traditional approaches to marketing, advertising, and branding is falling.

Mass-audience broadcasts are becoming less effective as individuals across the globe are increasingly able to share their interests or concerns, to benefit from the experiences of their peers, and to access expert knowledge. Consumer-created reviews and content now influences our spending patterns and compete for attention with the marketing messages distributed through traditional channels.

Traditional businesses are targeting the long tail niche markets for growth and finding themselves in competition with niche providers.

As the cost of customizing, personalizing, distributing, and accessing products, content, and services falls, it is becoming possible to penetrate niche markets that previously could not be serviced, were the exclusive domain was of niche providers or did not exist. The exploitation of these markets is described by the idea of the long tail economics, as they represent the vast number of markets that consist of small numbers of customers,

perhaps distributed geographically around the world. They are uneconomic to serve through traditional business models, but can be made accessible through the low-cost models enabled by Web 2.0 technologies. Their importance is increased by the saturation of traditional markets that has occurred in recent years through the sustained focus on business optimization in them. At the extreme of this scale is the phenomenon of peer-to-peer economic activity, where through the use of Internet-enabled brokers and mediators, individuals can leverage the communications, transactional, assurance, and distribution capabilities that have traditionally been the preserve of large corporations, to do business with each other directly.

- New user applications need to be delivered more rapidly than ever before, either to increase organizational agility or to deliver new products and services to market.

In some sectors, the commercially competitive life of products has reduced to the point where it is shorter than the average application deployment life cycle. So unless application delivery can be sped up beyond traditional limits, markets become commoditised before they can be reached. In other sectors, the need to rapidly respond to new situations or competitive threats means that business analysts, competitive experts, or operational decision makers need a new type of rapidly assembled data or a content-driven application to enable them to respond. For these, and other reasons, organizations are exploring the deployment and use of situational applications and enterprise mashups—effectively, the long tail of business solutions.

The three key software patterns (Figure 1-1) that are driving Web 2.0 are:

- ▶ **Service:** Data and logic are deployed into the Internet as services that can be easily invoked, upgraded, and deployed. This enables very low entry and high reuse.
- ▶ **Community:** Software is being developed as a community for the community using a variety of social networking tools. It enables quick feedback and comments.
- ▶ **Simple:** Massive push back on complexity, REST, and JSON are perfect examples of patterns and standards that provide enough of what most service developers need. Simple programming models, languages, run times, and specifications are driving Web 2.0 adoption by a wider audience than ever.

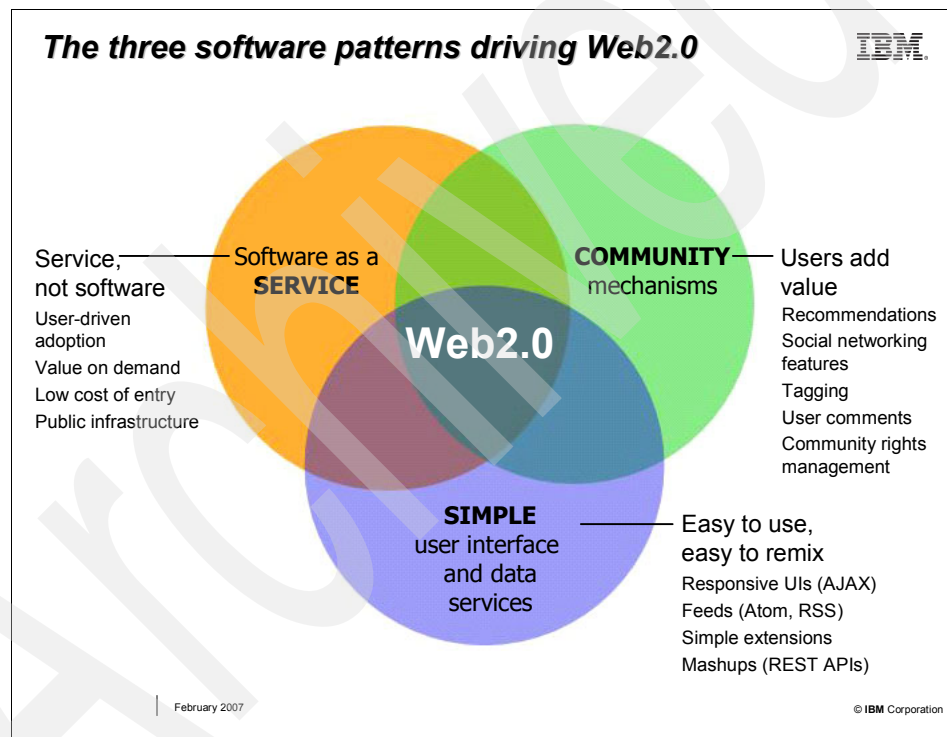


Figure 1-1 The three software patterns driving Web 2.0

## Summary

Web 2.0 is a mix of new approaches to interacting with customers, new Internet-centric business opportunities, and supporting technologies that let individuals connect and interact with each other more easily. In one sense it represents the emergence of new possibilities enabled by the widespread adoption of SOA principles and technologies. Web 2.0 offers business

opportunities, but brings challenges in how corporations embrace community, approach the sharing and protection of proprietary information, and identify and exploit the long tails in their marketplace.

## 1.4 Enterprise Web 2.0

Enterprise customers have already started to invest in SOA strategies, which allows the decoupling of the business services from the business processes. The next step is to understand which elements of these services and processes should be released in a form for internal integration through mashups or for release into the fabric of the Internet.

The key business asset in the Web 2.0 world is data, and enabling these feeds of data to be accessed in a RESTful way will enable enterprises to integrate themselves into new and innovative solutions. A good example of this is Google Maps. This has now become pervasive within the Internet and is used by millions of Web sites and developers. Google has managed to thread this service into the Internet and allow it to be consumed by a wide variety of Web 2.0 solutions.

The current set of SOA technologies in the form of Web services, WSDL, and SCA are focused on supporting computer-to-computer integration. They support a wide variety of capabilities that make these the correct solutions for enterprise integration within an Enterprise Service Bus architecture.

### **Web 2.0 is a pushback on complexity**

When it comes to becoming consumerable, WSDL and SOAP payloads are hard to create and require complex tooling and frameworks. The reason that Web 2.0 has gained such a wide adoption is that it allows developers and assemblers to go back to basics and focus on solutions to problems using easy-to-use concepts and technologies.

One of the reasons that SOAP and WSDL are not more widely adopted is because Web browsers do not natively support these two specifications. Most Web 2.0 solutions use the standards that are already included within the market-leading browsers already on consumer desktops, mobile computers, and mobile devices.

Enterprises must focus on how they can release their assets on the Internet. A key capability of the WebSphere Web 2.0 Feature Pack is that it can quickly enable assets already developed to be extended to support the protocols and formats that are being adopted within Web 2.0 solutions.

A key to staying competitive on the Internet is working out what services and data should be released for private or public consumption by your enterprise customers.

Figure 1-2 depicts many of the reasons why IBM has a core SOA strategy that focuses on Enterprise transactions and data. It is these assets that will be unlocked, exposed, and published onto the Internet. The following section describes techniques, using RESTful SOA, for exploiting these assets.

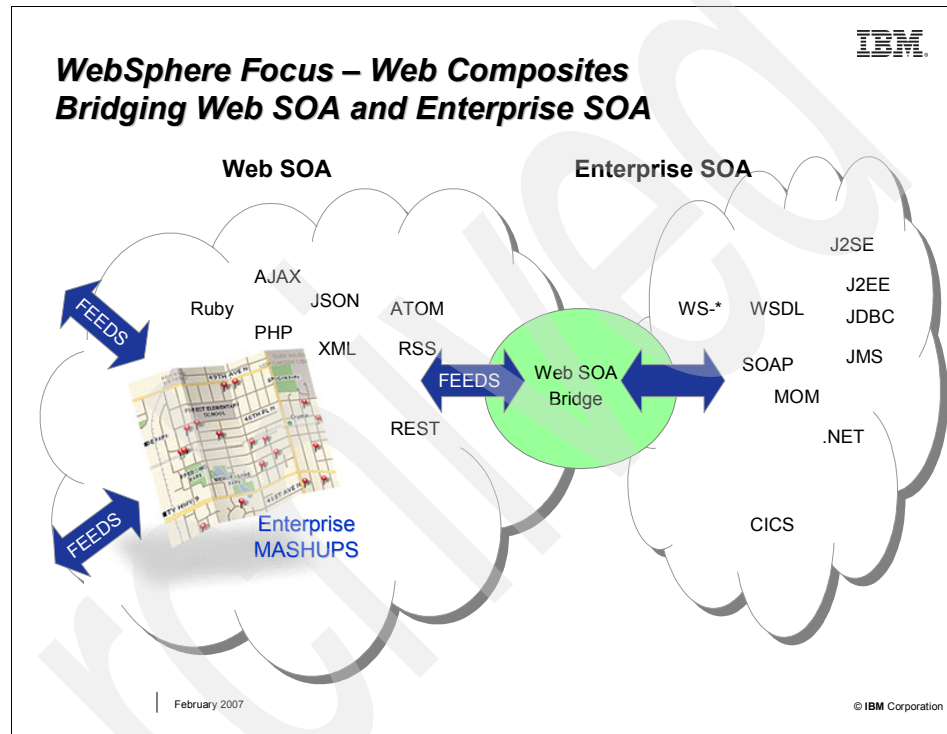


Figure 1-2 Enterprise Web 2.0 diagram

## 1.5 Extending the reach of SOA with RESTful SOA

We are in an interesting time right now for development of the Web. For the first time in several years we have a new set of technologies that let us look at Web development in a new and different way. Rather than viewing the Web as a net of entire pages, where Web-based applications generate a unique page at a time, we can now take a fundamentally different view of the Web. We can now view the Web as a set of services that can be asynchronously invoked, merged together with other services, and rewoven into new, dynamic combinations. This approach

is one of the defining characteristics of Web 2.0, and it is a significantly different approach to taking advantage of the Web.

The primary enabler of this new way of looking at the Web is a set of technologies commonly referred to as Ajax. Ajax a way of building Web pages that, rather than mixing content and presentation together, separates the two into a basic page that contains the presentation (lists, trees, and) and a set of XML documents that are asynchronously fetched (using JavaScript) and used to populate the presentation based on the actions of the user of the page.

This set of technologies has encouraged us to take a step back and think about what this means for enterprise computing. Where we previously considered that an enterprise services either ended at the corporate firewall, or at most was exposed under strict security to a limited set of corporate partners, we can now envision a time where some services extend all the way out to the browser. This is leading to a consideration of a new type of architecture, what we call a RESTful SOA. The way to view this is to take the basic idea of services extended all the way out to the browser, and then consider the impact that this has on the notion of an SOA.

RESTful SOA refers to delivering simple XML Web services delivered over HTTP using the representational state transfer approach. REST is an approach to developing services based on the foundation of the Web: HTTP. The next layer down is how you represent the information flowing from the service using technologies like ATOM feeds and JavaScript Object Notation (JSON). Finally, Ajax (which started this) is the way to render the information in the service to the user. However, the core idea remains that we are simply using a small set of protocols aimed at bringing information directly to the client.

### **1.5.1 REST and the foundation of the RESTful SOA**

As previously stated, REST is a key component in an implementation of a RESTful SOA. REST is a style of architecture that is best exemplified within the HTTP protocol. REST enables the creation of Web services that are simple to implement, built for re-use, and are ultra-scalable.

## Simple to implement

Creating a RESTful Web service is like forming a sentence. You need a noun, a verb, and an adjective. In REST terms, nouns are the resources pointed to by URLs. The verbs are the HTTP actions on resources. The adjectives are the formats of data in which the resources are represented. We like to lay this out in tables similar to the way we broke down sentences in grade school. For example, Table 1-1 lists how we would describe a set of services related to a photo album application.

Table 1-1 Photo album example usage of REST

Sentence (resource description)	Noun (URL)	Verb (action)	Adjectives (formats)
List all the albums.	.../album	GET	JSON
Show JPEG photo.	.../photo	GET	JPEG
Add a JPEG Photo.	.../photo	POST	JPEG
Delete a JPEG photo.	.../photo	DELETE	JPEG
List all the JPEG photos.	.../photo	GET	JPEG

Understanding this table is easy—each column breaks down into a different part of a sentence:

- ▶ Nouns/URLs: URLs are the most identifiable part of the Web, and as such straightforward way of organizing your services. Organizing a unique URL for each resource avoids confusion and promotes scalability.
- ▶ Verbs/actions: You can perform exactly four HTTP operations on these URLs:
  - POST
  - GET
  - PUT
  - DELETE

HTTP supports a few more actions, but these are the interesting ones for now. While perhaps constraining, the simplicity of these operations is somewhat liberating. These HTTP operations roughly map to the create, read, update, and delete (CRUD) operations that provide the foundational functions needed to implement a relational database. While simple, these four methods can be used in interesting and powerful ways.

- ▶ Adjectives/data formats: There are well known data types (MIME types, for example, text/html, image/jpeg) that are natively supported by HTTP servers

and browsers. Simple XML and JSON allow more custom data formats that are both self describing and can easily be parsed by the user. (When we say parse we also mean read with your eyes and parse with your brain.)

### **Built for reuse**

With the sentence metaphor as a basic building block, we can build on the RESTful service model and add new metaphors like feeds. The ATOM feed technology allows a user to create, update, and subscribe to resources. Restful feeds open up an interesting realm of possibilities. New feeds can be composed of existing feeds and tailored, merged, sorted, and filtered to provide a custom *skin* for your Web services.

### **Ultra-scalable**

The scalability of HTTP and the Web is well documented. The sentence approach to Web services allows standard HTTP server features to be used to optimize, secure, and scale your services. For example, existing HTTP caching proxies, firewalls, and routers can be effectively used with your application without requiring special knowledge of your application or implementation platform.

### **Why RESTful SOA is SOA**

Given the great benefits that can be derived from REST, the next question that comes to mind is, how does this fit into my enterprise? Cool technologies come and go, but if they do not fit in with existing systems then they are never going to make it out of the lab. This is where another recent architectural trend can help RESTful SOA find a home and turn it from a neat idea to a way to gain value in your business. We are referring to services-oriented architecture. SOA has emerged as the architecture of choice for building modern enterprise applications. The loosely coupled nature of SOA-based applications allows for business flexibility and reuse. Enterprise SOA provides a stringent set of capabilities and standards that allow mission-critical applications to be written using a SOA. Examples of technologies associated with enterprise SOA include WS\_\*, J2EE, and .NET. The IBM SOA home page defines SOA as:

A business-centric IT architectural approach that supports integrating your business as linked, repeatable business tasks, or services. SOA helps users build composite applications, which are applications that draw upon functionality from multiple sources within and beyond the enterprise to support horizontal business processes.

This is where the match comes in. SOA is all about providing easy access to the reusable business tasks in your enterprise (the services). It is also about combining all the different sources of information that you have inside and outside your enterprise to create new services. REST and the Web 2.0



movement, on the other hand, are about making these things available to your customers and users. The primary vehicle for this is what has been termed a mashup, defined as a Web application that combines data from more than one source into a single integrated tool. If you are thinking about building a mashup, then what better situation could you be in but having ready-made services providing the key business information in your enterprise available at your fingertips? Likewise, if you want your customers to provide you with content, then you want to do that in a way that is consistent with your business needs. Would it not be better if you controlled access to services like finding a store nearest a customer rather than having a customer scrape it off your Web site and get out-of-date information?

From this perspective, RESTful SOA is just a simplified instantiation of SOA that uses the Web as the SOA platform. You can look at RESTful SOA and compare it to your enterprise SOA by thinking about the different focuses of the two approaches. A primary focus of an Enterprise SOA is its flexibility and its ability to support a diverse and rich set of endpoints. The rich set of WS-\* standards that have been developed to deal with complex enterprise-level issues like security, transactions, and reliability are meant to promote interoperability between applications that could be written on widely diverse platforms. By comparison, a hallmark of a RESTful SOA is its simplicity and optimization for a very small set of clients and protocols.

Collectively, the RESTful SOA technologies can be used to create a radically simplified service platform using REST and ATOM to form the basis of the service invocation model, JSON and XML as the data interchange format, and Ajax as the model for a rich client. This platform can also be used to create an architectural *bridge* between RESTful SOA and enterprise SOA, allowing services from the enterprise to be simplified and reach the masses via the Web. We refer to this as *extending your SOA to the Web*.

## **Why you would want a RESTful SOA**

There are many benefits to a RESTful SOA. It allows you to build more interactive Web pages and enables your own customers to create new Web pages (or mashups) from your published services and those of others. It also opens up the notion of user-created content and allows you to contact your customers in new ways, such as blogs and wikis.

At its core, a RESTful SOA takes advantage of the most beneficial attributes of the World Wide Web. That is, it is pervasive, intuitive, and abundant. A RESTful SOA is a very cost-effective (in terms of both time and money) way to implement a SOA and extend the reach of your enterprise to consumers.

A RESTful SOA allows you to reach people who have so far been untouched by the SOA revolution. There are three primary ways to do that:

- ▶ Embracing the power of your business communities
- ▶ Unleashing your assets
- ▶ Extending your business processes to your communities

## **Embrace the power of your business communities**

Especially in a contracting economy, every small advantage you can gain over your competition is important. If you can reach your target market even a fraction of a percent better than your competition, and can successfully turn that reach into sales, then your investment will pay off. So, traditionally, businesses have used armies of sales people putting their feet onto the street in order to meet with customers and tell the story of their products. Likewise, traditional advertising venues like radio, TV, newspapers, magazines, and even standard Internet sites are simply a way to put a vision of your product in front of your potential customers. But there is another form of communication that can be even more effective at getting the word across, and even better targeted at the appropriate audience: word of mouth.

Every business has a set of communities around it. Your customers form a community. Your business partners form another community. Even your employees form a special community that you also can take advantage of: If they did not believe in your products, they would not work for you. In order to take advantage of word of mouth you must embrace your business communities. In the new age of social-software and community-based techniques, this is the most effective way to take advantage of the communities that you already have, and use them to drive sales, revenues, and new ideas.

So what does it mean to embrace your business communities? First of all, you have to open the communication channels to them, and more importantly, between them. Do not try to set the agenda of what they will communicate about, let the community set its own agenda. In practical terms, this means that as new Web sites and internal and external applications are developed, they should be supported by a set of mechanisms that support the user community in helping each other. This could be through a wiki, a forum, or other feedback mechanisms directly built into the applications themselves. Tim O'Reilly calls this *harnessing collective intelligence*.

One of the more interesting experiments that IBM has tried in the last several years is hosting the documentation for a few of its products, particularly WebSphere Extended Deployment (XD), on a wiki that is available for modification by any of the XD developers who are not just producers of the documentation, but an important group of consumers of the documentation as well. The key to building value in a social network is to let individual's decisions

create value as a by-product. That is what happens on the XD wiki. As a developer decides that they need more information about a certain topic, they expand the wiki. This is the principle behind Wikipedia, and it can be applied successfully to many types of documents besides encyclopedias and product documentation.

IBM has a number of different approaches to help you embrace your communities:

- ▶ Lotus Quickr provides a number of different approaches like teamplaces that help you connect your community members to each other.
- ▶ Project Zero will support a number of open source forums and other connectivity projects written in PHP.
- ▶ Research efforts like QEDWiki can even show how to write your applications directly into the wiki environment and blur the distinction between community-supported and community-created applications.

## 1.5.2 Unleash your assets

Let us assume that our previous argument is true and that RESTful SOA technologies are changing the way things work. If technologies like Ajax, REST, and ATOM can make it possible to reach your customers, your business partners, and even your own employees in a new and more effective way, then how do we enable the creation of applications that take advantage of these technologies? The next step is to unleash your assets.

Every IT shop has a number of internal assets that contain information and that perform functions that could be leveraged in new and innovative ways to create new business value. These could be:

- ▶ Database tables
- ▶ Web services
- ▶ Legacy transactions
- ▶ Third-party software functions

For example, consider the following situation. Assume that you are working in the IT shop of a large U.S. bank. What are some examples of data and functions that you could unleash to the Web? Well, let us assume that you have a database of bank locations and ATM locations throughout the country. If this was exposed through a REST interface, then it could be mashed up with Google Maps to display the locations on the map. This could lead a customer who is a fanatic about yard sales to build his own Google Maps mashup that shows the location of yard sales taken from the local newspaper's Web site and combines them with the nearest ATM locations, driving new business to your ATMs. Or imagine that you provide an ATOM feed of your new marketing initiatives (like if you invite a

friend to sign up for a new savings account, you receive \$20 in your account). Someone who runs a news aggregation site for frugal savers and coupon clippers could pick up on that feed and add it to their site, thus providing even more exposure for your new initiative.

The key here is to RESTfully enable everything. That does not mean that everything will be made visible outside of your firewall. Privacy and security concerns would prevent that. However, integration is just as important inside your firewall as it is outside your firewall. For instance, you would most likely never release your customer data outside your firewall. However, if you exposed your customer contact database as REST services inside your firewall it might be possible to better enable your salesmen to view their customer contacts geographically and plan trips that visit several at once. It also may also make it possible to reveal relationships and contacts within your customer base that you would never have seen by viewing a paper report.

It is a cumulative effect. As more of your assets are RESTfully enabled, even more value can be created by combining these previously separate islands of data or functions in new ways. This leads to faster integration of new data and applications into your existing Web sites, and allows you to take better advantage of the possibilities inherent in the rich Internet application approach.

### 1.5.3 Unleashing your J2EE assets

You can start to unleash you existing J2EE assets with the WebSphere Application Server Feature Pack for Web 2.0.

One of the critical issues new adopters of Ajax face is the potential of security holes caused by cross-domain scripting attacks when combining internal and external services. WebSphere Application Server Feature Pack for Web 2.0 includes an Ajax proxy component to address that concern. The feature pack also contains JSON libraries and Web remoting capability to make it easier to expose JEE Web services as REST services. The feature pack also provides ATOM feed syndication in order to provide another route for extending existing services outside the enterprise.

Finally, you want to extend the information available in your enterprise directly to your customers. The feature pack's Ajax Messaging component enables a publish and subscribe model whereby the server can stream data updates, messages, and events in real time to the client without the need for additional client-side plug-ins. This is implemented through server-side and client-side Ajax components that communicate with JSON-based messages. This means that information available from your Enterprise Service Bus (ESB) can be delivered dynamically to Internet clients.

## 1.5.4 Classification of feeds

With RESTful SOA becoming a key component to the WebSphere brand's product strategy, you will see other products across the software group portfolio becoming enabled to support restful SOA. You can start to think of these products as fitting into three main categories of enablement:

- ▶ Feed managers and producers
- ▶ Feed aggregator's
- ▶ Feed consumers

Each of these categorizations can help easily identify where the RESTful service endpoint can be used and consumed. If we look at feed managers and producers these are software group products like DB2 or CICS® that will enable data to be easily added into its restful endpoints in a create, read, and update pattern. If you look at the Lotus brand the current Lotus Connections product is an ATOM-based RESTful feed manager that allows data to be easily managed within its architecture. Tivoli® and its broad set of management products along with the WebSphere Portal product will open up to becoming feed managers, allowing a very easy entry point for usage within administration tasks.

Feed aggregating products will support the ability to handle the aggregation and mediation of feeds into composite feeds or to produce data for feed managers. These types of products are essential for helping organization prepare feeds for consumption by feed-consuming products. If you look across the software group you will see Project Zero and Information Management Info 2.0 products supporting RESTful SOA. In the future, Tivoli and WebSphere Message Broker will support the ability to aggregate feed content. Where you may have seen programmatic mediation in the past this could be opened up to Sayson'sistent set of feed aggregating tools or run times that support RESTful SOA.

Feed consumers are key to enabling the pervasive way that Web 2.0 and RESTful SOA have threaded themselves into the fabric of the Internet starting with the RSS/ATOM readers in the Web browser to the fully functional feed consumption support in Lotus Mash-up maker and Information Management Info 2.0 products. Feed consumers can manifest themselves as simple user products that live on the Internet or as SaaS products, or they could be easy-to-install desktop/device software, but when you analyze most Web 2.0 middleware solutions in the near future these will consume feeds directly in programmatic code or through scripting languages like Groovy support in Project Zero.

This simple categorisation will enable architects to thread IBM Software into a wide range of solutions using the common denominator of RESTful SOA as a key enabling technology.

## 1.6 What is Ajax

Asynchronous JavaScript and XML (Ajax) is a group of related Web development patterns and techniques that can be used to create interactive Web applications. These techniques give the user of a Web application the characteristic of increased responsiveness and interactivity with the Web page content. This is achieved by exchanging small amounts of data with the server asynchronously using JavaScript to communicate with the server. This means that the entire Web page is not reloaded in a request/response fashion. This is intended to increase the Web page's interactivity, speed, functionality, and usability.

Ajax is asynchronous, in that extra data is requested from the server and loaded in the background without interfering with the display and behavior of the existing page. JavaScript is the scripting language in which Ajax function calls are usually made. Data is retrieved using the XMLHttpRequest object that is available to scripting languages run in most current browsers, or, alternatively, through the use of Remote Scripting in browsers that do not support XMLHttpRequest. In any case, it is not required that the asynchronous content be formatted in XML.

Ajax is a cross-platform technique usable on many different operating systems, computer architectures, and Web browsers, as it is based on open standards such as JavaScript and the DOM.

### 1.6.1 History

The XMLHttpRequest concept was originally developed by Microsoft® as part of Outlook® Web Access 2000 as a server-side API call. At the time, it was not a standards-based Web client feature. However, de facto support for it was implemented by many major Web browsers. The Microsoft implementation is called XMLHttpRequest. It has been available since the introduction of Internet Explorer® 5.0[5] and is accessible via JScript®, VBScript, and other scripting languages supported by IE browsers.

The Mozilla project incorporated the first compatible native implementation of XMLHttpRequest in Mozilla 1.0 in 2002. This implementation was later followed by Apple since Safari 1.2, Konqueror, Opera Software since Opera 8.0 and iCab since 3.0b352.

The World Wide Web Consortium published a Working Draft specification for the XMLHttpRequest object's API on 5 April 2006. Its goal is “to document a minimum set of interoperable features based on existing implementations, allowing Web developers to use these features without platform-specific code.” The draft specification is based on existing popular implementations to help

improve and ensure interoperability of code across Web platforms. As of December 2007 the W3C standard was still a work in progress.

Web pages that use XMLHttpRequest or XMLHttpRequest can mitigate the current minor differences in the implementations either by encapsulating the XMLHttpRequest object in a JavaScript wrapper or by using an existing framework that does so. In either case, the wrapper should detect the abilities of the current implementation and work within its requirements.

Traditionally, there have been other methods to achieve a similar effect of server dynamic applications using scripting languages and plug-in technology.

## 1.6.2 Justification

The main justification for Ajax-style programming is to overcome the page loading requirements of HTML/HTTP-mediated Web pages. Ajax creates the necessary initial conditions for the evolution of complex, intuitive, dynamic, data-centric user interfaces in Web pages. Indeed, much of the Web's innovation and evolution during the Web 2.0 era has relied upon and benefited immensely from the capabilities of an Ajax platform.

Web pages, unlike native applications, are loosely coupled, meaning that the data that they display are not tightly bound to data sources and must be first marshaled (set out in proper order) into an HTML page format before they can be presented to a user agent on the client machine. For this reason, Web pages have to be re-loaded each time a user needs to view different data sets. By using the XMLHttpRequest object to request and return data without a re-load, a developer bypasses this requirement and makes the loosely coupled Web page behave much like a tightly coupled application, however, with a more variable lag time for the data to pass through a longer wire to the remote Web browser.

For example, in a classic desktop application, a Web developer has the option of populating a tree view control with all the data needed when the form initially loads, or with just the top-most level of data, which would load more quickly, especially when the data set is very large. In the second case, the application would fetch additional data into the tree control depending on which item the user selects. This functionality is difficult to achieve in a Web page without Ajax. To update the tree based on a user's selection would require the entire page to re-load, leading to a very jerky, non-intuitive feel for the Web user who is browsing the data in the tree.





## Feature pack overview

In this chapter we introduce you to various WebSphere Application Server Feature Packs available and features offered. Then we discuss the Feature Pack for Web 2.0 features, server-side libraries, connectivity protocols supported, and samples provided with Feature Pack for Web 2.0 and conclude with benefits of implementing this feature pack.

This chapter contains the following sections:

- ▶ Feature packs for WebSphere Application Server
- ▶ Feature Pack for Web 2.0 features
- ▶ Benefits of Feature Pack for Web 2.0

## 2.1 Feature packs for WebSphere Application Server

WebSphere Application Server feature packs are a mechanism for providing major new application server function and support for industry standards between product releases. These are helpful in quickly exploring and implementing new technologies within your business applications in today's rapidly changing business environments.

WebSphere Application Server feature packs are optionally installable product extensions that offer targeted, incremental new features. WebSphere Application Server customers who wish to take advantage of these features can download the appropriate feature pack and install it on their entitled application servers.

The primary characteristics of feature packs are:

- ▶ Feature packs provide production-ready function (such a new open standards) to customers who need them, without having to wait for a new WebSphere Application Server release.
- ▶ Customers can choose which feature packs, if any, they wish to install. In some cases, one feature pack may rely on the capabilities of another feature pack, in which case both must be installed.

Some of the recommended, generally available (GA) feature packs available are Feature Pack for Web 2.0, Feature Pack for Web Services, and Feature Pack for EJB™ 3.0.

### Feature Pack for Web 2.0

IBM WebSphere Application Server Feature Pack for Web 2.0 is for creating Ajax-based applications and mashups on WebSphere Application Server Versions 6.1 and 6.0.2 and for WebSphere Application Server Community Edition V2.0. The feature pack's functionality is intended to provide developers and architects with resources to create Ajax-styled Web applications and architectures, and includes both client-side runtime and server-side functionality.

Feature Pack for Web 2.0 highlights (Figure 2-1) are:

- ▶ **Web 2.0 to SOA connectivity:** For enabling connectivity from Ajax clients and mashups to external Web services, internal SOA services, and JEE assets. This extends enterprise data to customers and partners through Web feeds.
- ▶ **Ajax messaging:** For connecting Ajax clients to real-time updated data like stock quotes or instant messaging.
- ▶ **Ajax Development Toolkit:** Best-in-class Ajax development toolkit for WebSphere Application Server based on Dojo (<http://www.dojotoolkit.org>) with IBM extensions.

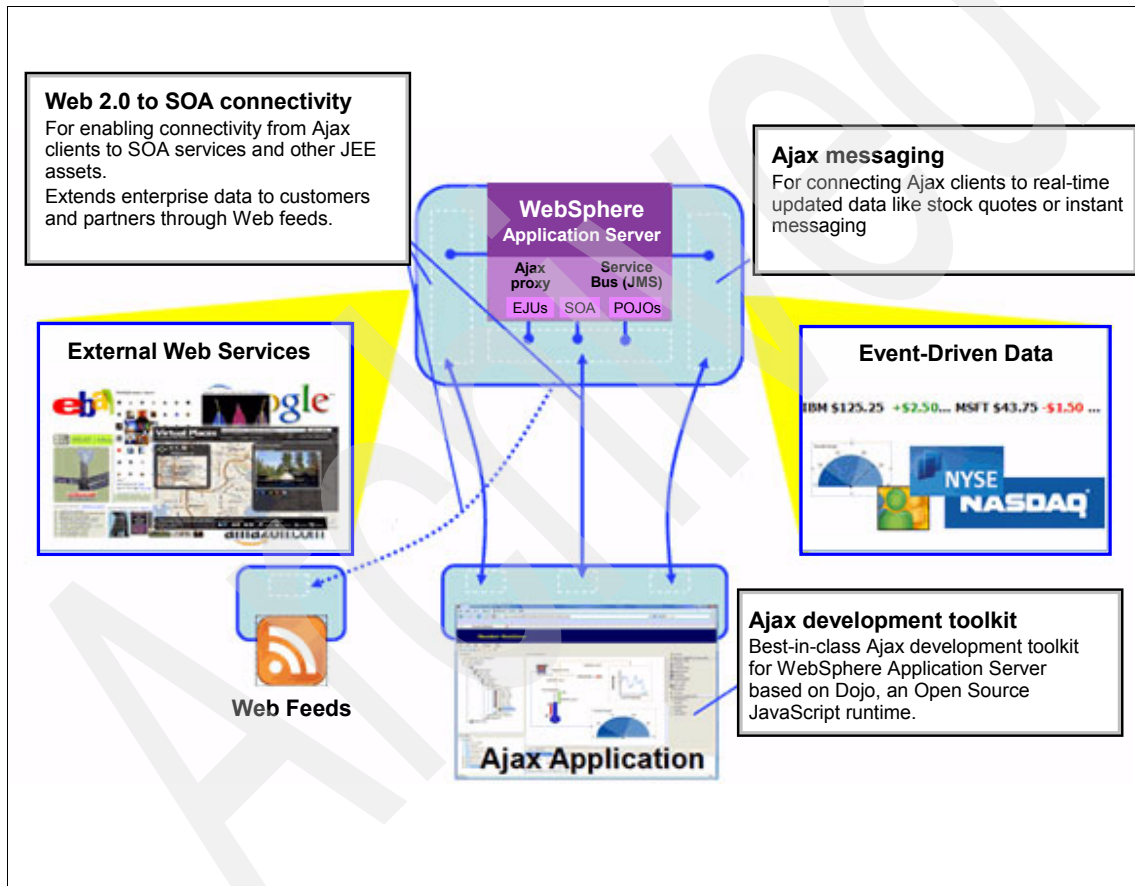


Figure 2-1 Feature Pack for Web 2.0 overview

## 2.2 Feature Pack for Web 2.0 features

In this section we discuss Feature Pack for Web 2.0 features.

### 2.2.1 Ajax connectivity

In this section we discuss Ajax connectivity.

#### **Remote Procedure Call Adapter (Web-remoting)**

Web-remoting is a pattern that provides support for JavaScript or client-side code to directly invoke server-side logic. This pattern provides the ability to invoke Java methods from JavaScript.

IBM implementation for Web remoting is referred to as the Remote Procedure Call (RPC) adapter for IBM. The RPC Adapter is designed to help developers create command-based services quickly and easily in a manner that complements programming styles for Ajax applications and other lightweight clients. Implemented as a generic servlet, the RPC Adapter provides an HTTP interface to registered JavaBeans™.

#### **Ajax proxy**

In a network environment, a proxy is positioned between the requesting client and the server. The proxy accepts requests from the client and passes them onto the server and brokers the response back to the client.

The Ajax proxy provided with IBM WebSphere Application Server Feature Pack for Web 2.0 is a reverse proxy. You can install a reverse proxy near one or more servers. Connections coming through the reverse proxy are forwarded to the requested server. From the client's perspective, the requests appear to originate from the same server, even though the reverse proxy might be forwarding requests to multiple Web servers.

The proxy can be used to broker client requests from multiple domains while using Ajax. JavaScript sandboxing rules prevent the start of network requests to servers from where the JavaScript did not originate. The policy is referred to as the same-origin policy on most major browsers. As an example, if the JavaScript application originated from domain A and attempts to use an XMLHttpRequest to domain B, then the browser prevents the domain B request. The proxy can be used to broker the request. From the client view, the request seems to come from the same server from which the JavaScript originated.

## Feeds library

Feeds are a mechanism to deliver content in a standardized specification format. Atom Syndication Format and RSS are two such specifications.

IBM support for feeds using Apache Abdera libraries is referred to as the feeds library. Apache Abdera addresses both the Atom syndication format and the Atom publishing protocol. In addition, Abdera currently supports only reading or parsing of RSS content.

## 2.2.2 Server-side libraries

In this section we discuss server-side libraries.

### JSON4J

The JavaScript Object Notation (JSON4J) library is an implementation of a set of JavaScript Object Notation (JSON) handling classes for use within Java environments. The JSON4J library provides the following functions:

- ▶ A simple Java model for constructing and manipulating data to be rendered as the JSON implementation.
- ▶ A fast transform for XML to JSON conversion for situations where conversion from an XML reply from a Web service into a JSON structure is desired for easy use in Ajax applications.
- ▶ A JSON string and stream parser that can generate the corresponding JSONObject, which represents that JSON structure in Java. You can then make changes to that JSONObject and serialize the changes back to the JSON implementation.

### Ajax messaging

The Ajax messaging service is a publish and subscribe implementation that connects the browser to the WebSphere Application Server service integration bus (SIBus) for server-side event push.

Client/server communication is achieved through the Bayeux protocol. The Bayeux protocol is an HTTP-based messaging routing protocol. Client support for the Bayeux protocol is provided by the Dojo Toolkit. The Web message service implementation bridges incoming Bayeux requests to the service integration bus allowing Web services, JMS clients, or any item connected to the service integration bus to publish events to Web-based clients.

## 2.2.3 Ajax client runtime (Ajax Development Toolkit)

The Ajax client runtime is a powerful Ajax development toolkit that is based on an IBM supported version of the Dojo Toolkit Version 1.0.

The Ajax client runtime includes extensions to the Dojo Toolkit that provide support for the areas discussed below.

### SOAP library

SOAP connectivity makes it easier to invoke public SOAP-based Web services from Ajax applications.

- ▶ The SOAP service: This library extends the `dojo.rpc.RpcService` class and provides an easier way to create the SOAP envelope around a request.
- ▶ The SOAP widget: This widget uses the SOAP service and enables a convenient way to connect to external SOAP services and invoke their methods in a simple way.

### Atom library

The Atom Publishing Protocol (APP) data access makes it easy to invoke any Atom APP 1.0-compliant service, and uses Atom feeds as data sources that are bound to widgets within an Ajax application. Atom feed widgets are sample feed widgets that use the Atom APP data access feature.

- ▶ The Atom library  
This library contains three different features. First are the general utility functions to support the rest of the library. Next is the data model for the various parts of Atom, such as content, person, link, feed, and entry. These data models are then used to define the `AtomIO` object, which is a wrapper to the various functions intrinsic to Atom feeds and the Atom Publishing Protocol.
- ▶ The AppStore  
Implementing the `dojo.data.api` API Read, Identity, and Write APIs, the AppStore handles reading and writing from an APP source in a implementation agnostic way. It also supports one to fetch and store entries without knowing about the APP underpinnings.

► The Atom widgets

These are sample feed widgets that use the Atom (APP) data access feature. Each of these three widgets performs a different function:

- The *feed viewer widget* is a reference widget for a Atom feed viewer implementation. This widget fetches a feed from the given URL and displays, in chronological order, entry titles and updated elements. This widget supports click events to highlight an entry and publishes the selected entry on the given topic.
- The *feed entry viewer widget* displays individual entry elements. A bar is provided at the top with a menu to turn specific elements on or off. The viewable elements can be dictated at creation time. In addition to the functions outlined, there are several additional functions, used mainly internally, to set the value of various nodes in the page.
- The *feed entry editor widget* displays the elements of an individual entry. However, it also allows edits to be made to this entry, if the entry has a link with a real attribute that equals to *edit*. This widget does none of the work of sending the edits back as requests. Instead, the widget updates its rendering and sends the updates on the `entrySelectionTopic` and allows the `FeedViewer` widget to handle the sending of the data. This widget extends the `FeedViewer` widget. Therefore, all of its variables and functions are inherited. This widget does, however, override the previously mentioned undocumented functions that set the display node values, as it must replace the value with an editor object in the event that this entry is being edited. This widget has no additional variables or functions over the `FeedEntryViewer`.

► Gauge widgets

Gauge widgets are a way to display numerical data graphically. There are two Gauge widgets:

- Analog Gauge widget
- Bar Graph widget

These widgets support real-time updates (for example, when used with Ajax messaging service).

► OpenSearch Library

An OpenSearch Data Store, which is an implementation of the `dojo.data` API's OpenSearch library, makes it easy to invoke any OpenSearch-compliant service and bind the search results to widgets within an Ajax application. Typically, server-side support consists of hosting an open search description document that defines the URLs used to query the server. When the store is instantiated, it parses the description document and determines the best URL element to use. The types, in order, are Atom, RSS, and HTML. The store queries the server to retrieve the results.

## 2.2.4 Web 2.0 sample applications

In this section we discuss Web 2.0 sample applications.

### Quote Streamer sample

The quote streamer sample application uses the Web messaging service to simulate stock quotes being delivered to a Dojo-enabled client application. The Web messaging service links a Dojo-enabled client, a WebSphere Application Server internal message broker, and platform messaging for Web-based publication or subscription.

In the quote streamer sample application, multiple Dojo widgets are specified in a market report summary article in HTML format. These Dojo widgets process incoming stock quote messages and visually indicate stock changes. These visual changes include:

- ▶ Updates to the current price of a stock
- ▶ Daily stock price change
- ▶ Daily stock percent price change

When the price of a stock changes, the daily price change and daily percent price change represent green (increase) or red (decrease) before returning to the original background color.

### Plants by WebSphere sample

The Plants By WebSphere application is a fictional Web site that makes available nursery items such as vegetables, flowers, accessories, and trees. You can view an online catalog, select items, and add them to a cart. When the cart contains items, you can proceed to log in, supply credit card information, and check out.

This application contains model, presentation, and controller layers that can be developed. Additionally, Ajax-style architecture is added through the JavaScript Dojo Toolkit. Specifically, the use of various Dojo widgets and enablement of drag-and-drop capabilities for the shopping cart are added.

### RPC Adapter sample

The CourierApp sample Web application is a sample that demonstrates the capabilities of the RPC Adapter such as handling of XML and JSON responses, white and black list methods, and specifying validators. The CourierApp sample application demonstrates the auto population of address details upon the selecting of a zip or postal code. The RPC Adapter's JSON service is used to access the address lookup service. The same application is also implemented using the RPC Adapter's XML RPC.



## 2.2.5 FeedSphere sample

The FeedSphere sample demonstrates the ATOM and RSS support in Abdera. The specific scenarios covered are:

- ▶ Generate an ATOM feed: A simple example to create an ATOM feed.
- ▶ Read an ATOM feed: An example of reading ATOM feed content, as well as performing a filtered read on the ATOM feed content.
- ▶ Read an RSS feed: An example of reading RSS feed content.
- ▶ ATOM Publishing Protocol support: Demonstrates the ability to create, update, delete, and retrieve entries in a feed. Also provides options to retrieve the service document for the deployed Abdera server and the associated feed.

## 2.3 Benefits of Feature Pack for Web 2.0

The feature pack provides WebSphere Application Server customers with the ability to enable more interactive, dynamic Web applications and complete leverage on existing SOA and J2EE assets by making them easily accessible to Web 2.0 and Ajax applications. The feature pack uses WebSphere Application Server to manage cross-site access to Web 2.0 style services from browser-based Ajax applications.

Feature Pack for Web 2.0 can be helpful in reducing development costs and time to market with Ajax enhancements to J2EE applications by leveraging existing Java development skills, J2EE object reuse, and IBM support.

## 2.4 References

For additional information see:

- ▶ WebSphere Information Center  
<http://publib.boulder.ibm.com/infocenter/wasinfo/v6r0/index.jsp>
- ▶ IBM Education Assistant: WebSphere  
<http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp>
- ▶ IBM Developer Works: WebSphere  
<http://www.ibm.com/developerworks/websphere>



## Core concepts

This chapter outlines the central technologies and core concepts for Web development with IBM WebSphere Feature Pack for Web 2.0. The following concepts are presented:

- ▶ HTTP Protocol
- ▶ REST
- ▶ Asynchronous JavaScript (Ajax)
- ▶ JSON
- ▶ POX (XML)
- ▶ Atom
- ▶ JavaScript
- ▶ DOM
- ▶ DHTML
- ▶ CSS
- ▶ Web 2.0

## 3.1 The technical foundations of Web 2.0

To enable the next generation of Web 2.0-based solutions to be delivered with the WebSphere Application Server, the Web 2.0 Feature Pack has been released to support the core technical elements required for Web 2.0 application development.

This chapter explains the technical foundations of a modern Web 2.0 application and provides explanations and samples of these technologies, enabling the reader to obtain a clearer understanding of the technical concepts that are used throughout this book.

Not all applications will use all these technology elements or patterns. It will be important for the developer to produce a clear design of how the various tiers of the new application will be constructed and how these tiers will communicate with each other. With Web 2.0 the Model View Controller (MVC) pattern is still evident within the architecture. However, the Controller function is now moved to be within the browser and it is the browser's responsibility to manage the invocation of data services within the server tier. Once the request is within the server tier it can use a variety of technologies to retrieve data through persistence, messaging, or Web services. See Figure 3-1.

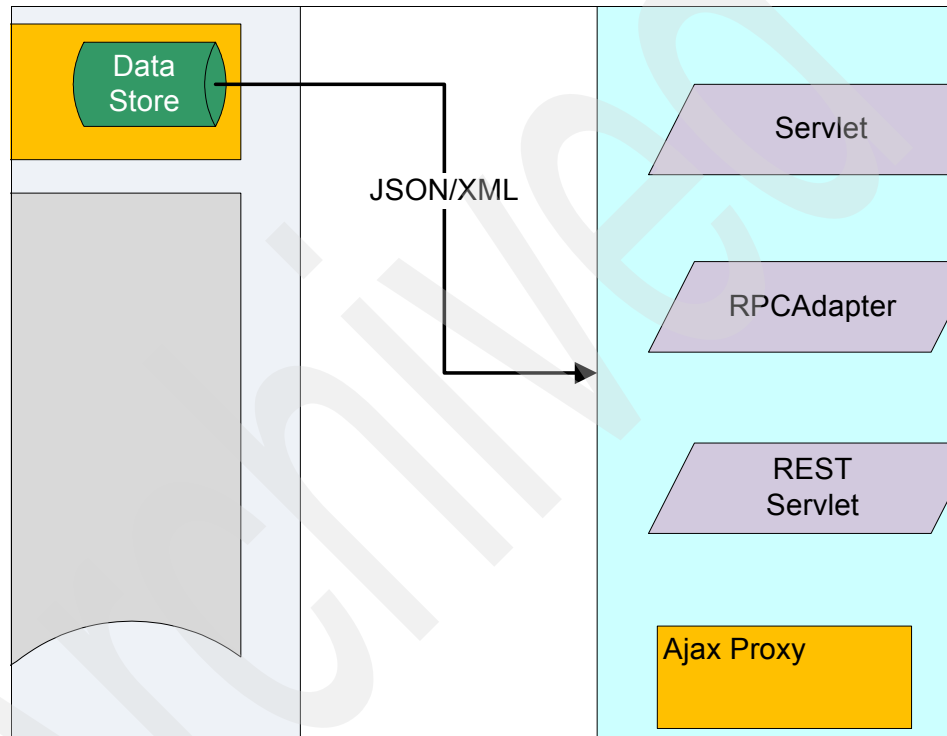


Figure 3-1 Technology tiers for a modern Web 2.0 application

The following sections explain the technologies that form the Web 2.0 technical stack.

### 3.1.1 HTTP Protocol

Hypertext Transfer Protocol (HTTP) is a communications protocol for the transfer of information on intranets and the World Wide Web. Its original purpose was to provide a way to publish and retrieve hypertext pages over the Internet.

HTTP development was coordinated by the World Wide Web Consortium (W3C) and the Internet Engineering Task Force (IETF), culminating in the publication of a series of Request for Comments (RFCs), most notably RFC 2616 (June 1999), which defines HTTP/1.1, the version of HTTP in common use.

HTTP is a request/response standard between a client and a server. A client is the user and the server is the Web site. The client making an HTTP request using a Web browser, spider, or other user tool is referred to as the user agent. The responding server, which stores or creates resources such as HTML files and images, is called the origin server. In between the user agent and the origin server may be several intermediaries, such as proxies, gateways, and tunnels. HTTP is not constrained to using TCP/IP and its supporting layers, although TCP/IP is the most popular transport mechanism on the Internet. Indeed, HTTP can be implemented on top of any other protocol on the Internet, or on other networks. HTTP only presumes a reliable transport. Any protocol that provides such guarantees can be used.

Typically, an HTTP client initiates a request. It establishes a Transmission Control Protocol (TCP) connection to a particular port on a host (port 80 by default). An HTTP server listening on that port waits for the client to send a request message. Upon receiving the request, the server sends back a status line, such as HTTP/1.1 200 OK, and a message of its own, the body of which is perhaps the requested file, an error message, or some other information.

Resources to be accessed by HTTP are identified using Uniform Resource Identifiers (URIs) (or, more specifically, Uniform Resource Locators (URLs)) using the http or https URI schemes.

### 3.1.2 REST

Representational state transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. The terms *representational state transfer* and *REST* were introduced in 2000 in the doctoral dissertation of Roy Fielding, one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification. The terms have since come into widespread use in the networking community.

REST strictly refers to a collection of network architecture principles that outline how resources are defined and addressed. The term is often used in a looser sense to describe any simple interface that transmits domain-specific data over HTTP without an additional messaging layer such as SOAP or session tracking via HTTP cookies.

### 3.1.3 Asynchronous JavaScript (Ajax)

Ajax is a group of inter-related Web development techniques used for creating interactive Web applications. A primary characteristic is the increased responsiveness and interactivity of Web pages achieved by exchanging small amounts of data with the server *behind the scenes* so that entire Web pages do not have to be reloaded each time that there is a need to fetch data from the server. This is intended to increase the Web page's interactivity, speed, functionality, and usability.

Ajax is asynchronous, in that extra data is requested from the server and loaded in the background without interfering with the display and behavior of the existing page. JavaScript is the scripting language in which Ajax function calls are usually made. Data is retrieved using the XMLHttpRequest object that is available to scripting languages run in modern browsers or, alternatively, through the use of Remote Scripting in browsers that do not support XMLHttpRequest. In any case, it is not required that the asynchronous content be formatted in XML.

Ajax is a cross-platform technique usable on many different operating systems, computer architectures, and Web browsers, as it is based on open standards such as JavaScript and the DOM. There are free and open source implementations of suitable frameworks and libraries.

### 3.1.4 JSON

JavaScript Object Notation (JSON) is a lightweight computer data interchange format. It is a text-based, human-readable format for representing simple data structures and associative arrays (called objects).

The JSON format was specified in RFC 4627 by Douglas Crockford. The official Internet media type for JSON is application/json. The JSON file extension is .json.

The JSON format is often used for transmitting structured data over a network connection in a process called serialization. It is primarily used in Ajax Web application programming, where it serves as an alternative to the traditional use of the XML format.

Although JSON was based on a subset of the JavaScript programming language (specifically, Standard ECMA-262 3rd Edition—December 1999) and is commonly used with that language, it is considered to be a language-independent data format. Code for parsing and generating JSON data is readily available for a large variety of programming languages. The JSON Web site (<http://json.org/>) provides a comprehensive listing of existing JSON bindings, organized by programming language.

### 3.1.5 POX (XML)

Plain Old XML (POX) is a term used to describe basic XML, sometimes mixed in with other, blendable specifications like XML Namespaces, XInclude, and XLink. The term is typically used in contrast with complicated, multilayered XML specifications like those for Web services or RDF. The term may have been derived from or inspired by the expression Plain Old Java Object (that is, POJO).

POX is different from REST in that REST refers to a style for communication protocols, while POX only refers to an information format style.

The primary competitors to POX are syntax standards such as YAML and JSON-derivatives that do not utilize XML, in addition to structured protocols like SOAP.

### 3.1.6 Atom

The name Atom refers to a pair of related standards. The Atom Syndication Format is an XML language used for Web feeds, while the Atom Publishing Protocol (short AtomPub or APP) is a simple HTTP-based protocol for creating and updating Web-based applications.

Web feeds allow software programs to check for updates published on a Web site. To provide a Web feed, a site owner may use specialized software (such as a content management system) that publishes a list (or feed) of recent articles or content in a standardized, machine-readable format. The feed can then be downloaded by Web sites that syndicate content from the feed, or by feed reader programs that allow Internet users to subscribe to feeds and view their content.

A feed contains entries, which may be headlines, full-text articles, excerpts, summaries, or links to content on a Web site, along with various metadata.

As a response to the incompatibility between some of the versions of the RSS syndication format, the Atom format was developed as an alternative to RSS.

Proponents of the new format formed the IETF Atom Publishing Format and Protocol Workgroup. The Atom syndication format was published as an IETF proposed standard in RFC 4287, and the Atom Publishing Protocol was published as RFC 5023.

### 3.1.7 JavaScript

JavaScript is a scripting language most often used for client-side Web development. It was the originating dialect of the ECMAScript standard. It is a



dynamic, weakly typed, prototype-based language with first-class functions. JavaScript was influenced by many languages and was designed to appear like Java, but be easier for non-programmers to work with. The language is best known for its use in Web sites (as client-side JavaScript).

JavaScript, despite the name, is essentially unrelated to the Java programming language, though both have the common C syntax, and JavaScript copies many Java names and naming conventions.

JavaScript is a trademark of Sun™ Microsystems. It was used under license for technology invented and implemented by Netscape Communications and current entities such as the Mozilla Foundation.

### 3.1.8 DOM

The Document Object Model (DOM) is a platform-independent and language-independent standard object model for representing HTML or XML and related formats.

A Web browser is not obliged to use DOM in order to render an HTML document. However, the DOM is required by JavaScript scripts that wish to inspect or modify a Web page dynamically. In other words, the Document Object Model is the way that JavaScript views its containing HTML page and browser state.

Because the DOM supports navigation in any direction (for example, parent and previous sibling) and allows for arbitrary modifications, an implementation must at least buffer the document that has been read so far (or some parsed form of it). Hence, the DOM is likely to be best suited for applications where the document must be accessed repeatedly or in an out-of-sequence order.

### 3.1.9 DHTML

Dynamic HTML (DHTML) is a collection of technologies used together to create interactive and animated Web sites by using a combination of a static markup language (such as HTML), a client-side scripting language (such as JavaScript), a presentation definition language (Cascading Style Sheets, CSS), and the Document Object Model.

A DHTML Web page is any Web page on which client-side scripting changes variables of the presentation definition language, which in turn affects the look and function of the otherwise *static* HTML page content. However, the dynamic characteristic of DHTML is in the way that it functions while a page is viewed, not in its ability to generate a unique page with each page load.

### 3.1.10 CSS

Cascading Style Sheets (CSS) is a stylesheet language used to describe the presentation of a document written in a markup language. Its most common application is to style Web pages written in HTML and XHTML, but the language can be applied to any XML document.

CSS is used to help developers of Web pages define colors, fonts, layout, and other aspects of the document presentation. It is designed primarily to enable the separation of document content (written in HTML or a similar markup language) from document presentation (written in CSS). This separation can improve content accessibility, provide more flexibility and control in the specification of presentation characteristics, and reduce complexity and repetition in the structural content. CSS can also allow the same markup page to be presented in different styles for different rendering methods, such as on-screen, in print, by voice (when read out by a speech-based browser or screen reader), and on Braille-based, tactile devices. CSS specifies a priority scheme to determine which style rules apply if more than one rule matches against a particular element. In this so-called cascade, priorities or weights are calculated and assigned to rules so that the results are predictable.

The CSS specifications are maintained by the World Wide Web Consortium (W3C). Internet media type (MIME type) text/CSS is registered for use with CSS by RFC 2318.

## 3.2 Model View Controller

The Model View Controller (MVC) pattern can separate business logic from user interfaces, meaning that the underlying business logic or the application's presentation may be altered without affecting the other.

### 3.2.1 Model 2 J2EE MVC

Most developers of Web-based applications are familiar with the J2EE/JEE Model 2 design model, which utilizes the Model-View-Controller (MVC) pattern to separate the data content from the presentation. See Figure 3-2.

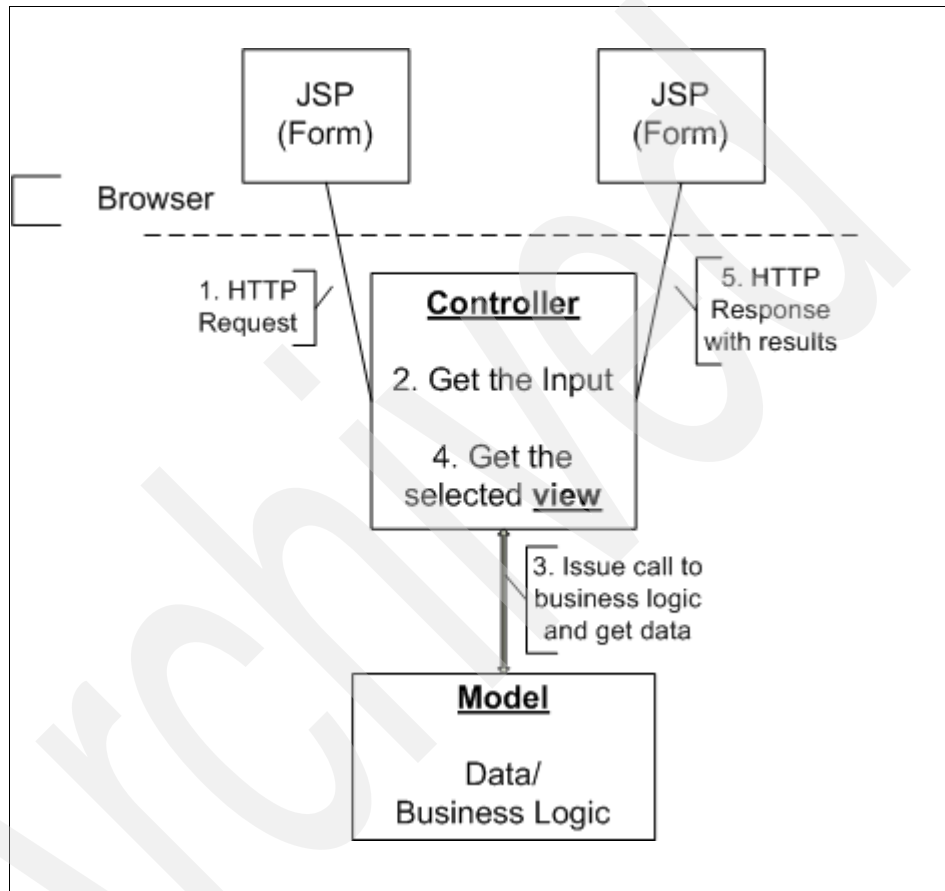


Figure 3-2 The J2EE Model 2 MVC pattern

This enterprise MVC implementation was traditionally structured with the controller being a servlet that receives input from the UI (for example, an input form from a JSP™), which calls business logic in the back-end server, which then forwards the result of the request to the appropriate JSP to be rendered. As the business logic can supply a diverse range of data, there is often a server-side application controller acting as a session facade to control invoking lower-level calls and to perhaps also build the response.

In terms of porting this setup to a Web service, this application controller could be exposed as an application-fronting servlet, responsible for accepting SOAP input, marshalling that input to the back-end service, and then returning the response in a defined mark-up language such as XML.

### **3.2.2 Web 2.0**

In this section we discuss Web 2.0.

#### **Ajax View and Controller in Browser**

With Ajax, a Web application can directly call those services from the browser. With the increased client interaction that normally is associated with a Web application, there will be far more requests for the server-side controller to render for the client's view, leading to bloat.

In the Web 2.0 model, however, the rendering (view) and controlling logic can be moved to reside in the browser. In this scenario the initial view is rendered in the traditional manner by the server, but after this the client can begin to take over the UI controlling logic.

## Client-side data store

To further reduce unnecessary, costly interaction with the server, the client can maintain a local data store for UI responsiveness, and certain UI functions will trigger persisting local data to the server. The Data API in the IBM Dojo Toolkit (Ajax client runtime) is an example of this. See 9.9, “Data API” on page 371, for more details. See Figure 3-3 for a Web 2.0 MVC example using IBM WebSphere Portal.

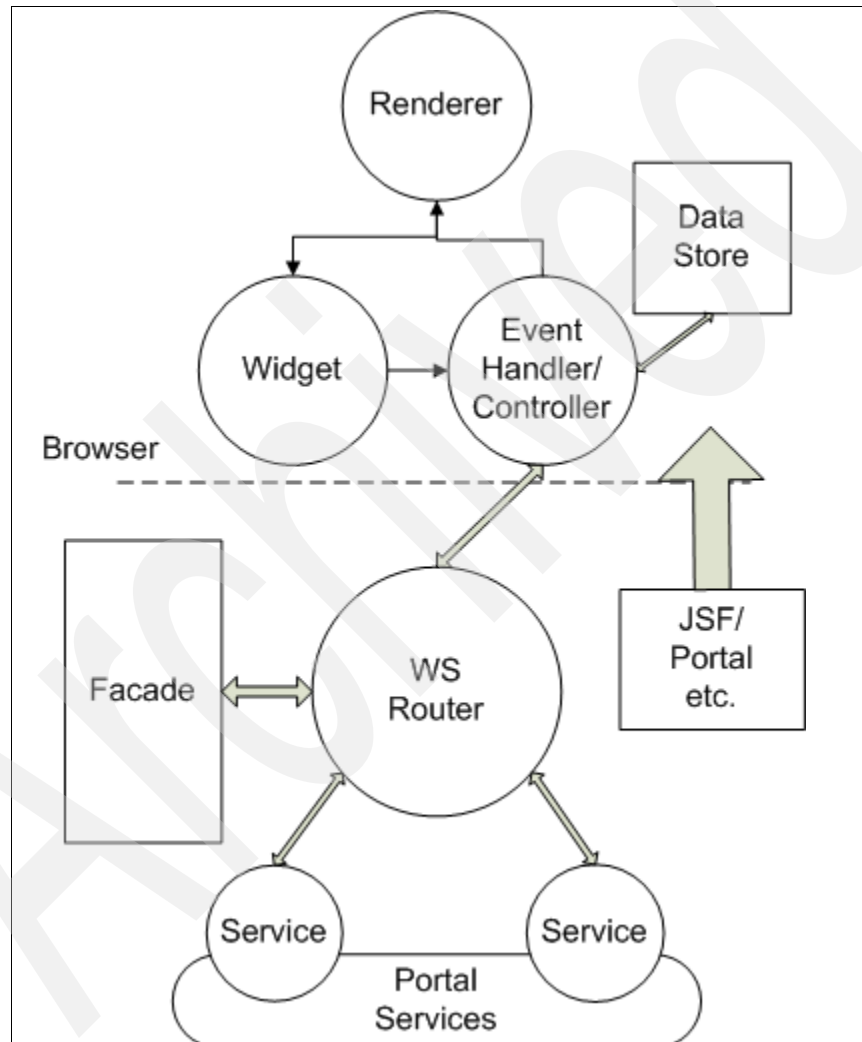


Figure 3-3 Example Web 2.0 MVC pattern using IBM WebSphere Portal



# Installation

This chapter explains the supported platforms and run times for Feature Pack for Web 2.0 and where to download, how to install, and how to uninstall it. The sections in this chapter are:

- ▶ Supported platforms
- ▶ Supported Integrated Development Environments (IDEs)
- ▶ Installing the application server product
- ▶ Downloading Feature Pack for Web 2.0
- ▶ Installing Feature Pack for Web 2.0
- ▶ Uninstalling Feature Pack for Web 2.0

## 4.1 Supported platforms

The following platforms are supported for Feature Pack for Web 2.0:

- ▶ AIX®
- ▶ HP-UX
- ▶ i5/OS®
- ▶ Linux®
- ▶ Linux on pSeries®
- ▶ Solaris™
- ▶ Windows®
- ▶ z/OS

## 4.2 Supported Integrated Development Environments (IDEs)

The following IDEs are supported for Feature Pack for Web 2.0:

- ▶ Rational Application Developer Version 7 with fix pack level 3 (7.0.0.3)
- ▶ Eclipse 3.3
- ▶ Eclipse plus Web tools platform (WTP) 1.5

## 4.3 Installing the application server product

**Note:** Find single operating system installation instructions under the various single operating system navigation options available in the IBM WebSphere Application Server Information Center.

Before Feature Pack for Web 2.0 can be installed, IBM WebSphere Application Server Version 6.x must first be installed. If the application server has been installed already, read no further on this topic and see 4.4, “Downloading Feature Pack for Web 2.0” on page 45. Otherwise, to install Feature Pack for Web 2.0:

1. Access the IBM WebSphere Application Server Information Center.
2. In the information center navigation, select the documentation for the product edition and single operating system required.
3. Expand the installation section.
4. Expand the installation instructions tailored to the single operating system required.



### 4.3.1 Updating the application server

Feature Pack for Web 2.0 requires the application server to be updated to the following versions:

- ▶ IBM WebSphere Application Server Version 6.0 requires at least version 6.0.2.23.
- ▶ IBM WebSphere Application Server Version 6.1 requires at least version 6.1.0.13.

### 4.3.2 Applying the appropriate iFix for the Web messaging feature

The Web messaging service is a publish and subscribe implementation that connects the browser to the WebSphere Application Server service integration bus (SIBus) for server-side event push. The following optional interim fixes provide support for using the Web messaging feature of Feature Pack for Web 2.0:

- ▶ PK56872; 6.0.2.23: Web 2.0 Feature Pack Web messaging function enablement fix
- ▶ PK56881; 6.1.0.13: Web 2.0 Feature Pack Web messaging enablement fix

**Tip:** Test the application server product installation before downloading and installing the Feature Pack for Web 2.0.

## 4.4 Downloading Feature Pack for Web 2.0

The Feature Pack for Web 2.0 can be downloaded from the following locations:

- ▶ The WebSphere Application Server Version 6.1 Feature Packs service page  
<http://www.ibm.com/software/Webservers/appserv/was/featurepacks>
- ▶ IBM Passport Advantage®  
<http://www.ibm.com/software/howtobuy/passportadvantage>

What is installed for the Feature Pack for Web 2.0 depends on the version of WebSphere Application Server running and the operating system.

### 4.4.1 WebSphere Application Server Community Edition Version 2.0

A stand-alone archive is provided:

- ▶ 2.0-WS-WASCE-WEB2FEP-MultiOS.zip
- ▶ 2.0-WS-WASCE-WEB2FEP-MultiOS.tar.gz

### 4.4.2 WebSphere Application Server Version 6 for Distributed, i5/OS

An installer for all the features and an eclipse-update-site.zip for easy import of features into Eclipse is provided:

- ▶ 6.x-WS-WAS-WEB2FEP-MultiOS.zip
- ▶ 6.x-WS-WAS-WEB2FEP-MultiOS.tar.gz

### 4.4.3 WebSphere Application Server Version 6 for z/OS

A stand-alone archive containing features applicable to z/OS is provided:

- ▶ 6.0.2-WS-WAS-WEB2FEP-ZOS.zip
- ▶ 6.0.2-WS-WAS-WEB2FEP-ZOS.tar.gz
- ▶ 6.1.0-WS-WAS-WEB2FEP-ZOS.zip
- ▶ 6.1.0-WS-WAS-WEB2FEP-ZOS.tar.gz

## 4.5 Installing Feature Pack for Web 2.0

This section outlines the installation instructions for installing Feature Pack for Web 2.0 for the appropriate operating system.

### 4.5.1 System requirements

The operating system requirements for Feature Pack for Web 2.0 are the same as for the application server installation. However, ensure that enough free disk space exists to install Feature Pack for Web 2.0 onto the application server installation. The following disk space is required:

- ▶ 280 MB of free space available in the target installation directory. This is in addition to the required application server product.
- ▶ 250 MB of free space available in the temporary directory.

## 4.5.2 Installing Feature Pack for Web 2.0 on distributed OSs

The following installation instructions describe how to install Feature Pack for Web 2.0 on distributed operating systems (OSs).

### Installing using the installation wizard graphical interface

To install:

1. Log on as the user who installed the application server product.
  - |     |       |       |         |
|-----|-------|-------|---------|
| AIX | HP-UX | Linux | Solaris |
|-----|-------|-------|---------|

Log on using the same user ID that was used when the product was installed.

Run the `ls -al` command at the root of the application server installation to find the user ID that was used to install the product.
  - |         |
|---------|
| Windows |
|---------|

Log on as a user who belongs to the administrator group or as the user who installed the product.
2. Stop the application server and any other processes for the product on which the feature pack is being installed. For example, for the Network Deployment product, stop all application server processes, the node agent, and the deployment manager.
3. Run the feature pack installation program.

Extract the downloaded product files and run the program manually from a command prompt. Run the install executable file in the root directory of the feature pack installation image and point to the Java Virtual Machine (JVM™) already included with the application server, where `<app_server_root>` is the root directory of the existing application server product:

  - |     |       |       |         |
|-----|-------|-------|---------|
| AIX | HP-UX | Linux | Solaris |
|-----|-------|-------|---------|

```
./install -is:javahome <app_server_root>/java
```
  - |         |
|---------|
| Windows |
|---------|

```
install -is:javahome <app_server_root>\java
```
4. Complete the steps in the feature pack installation wizard.
  - a. On the Welcome panel, click **Next**.
  - b. On the Software License Agreement panel, select **I accept both the IBM and the non-IBM terms** and click **Next**.
  - c. On the Installation Directory panel, specify the location of the `<app_server_root>` directory of the existing WebSphere Application Server installation and click **Next**.

- d. On the Installation Summary panel, click **Next**. The program copies feature pack program files to the existing application server installation.
- e. The installation completes and the Installation Results panel is displayed. If the application server is not at the required version (see 4.3.1, “Updating the application server” on page 45), a message is displayed indicating that the appropriate fix packs should be installed to reach the supported level before contacting IBM Support for any subsequent problems. Use the IBM Update Installer for WebSphere Software to update the application server to the required level.

Click **Finish**.

## Installing silently using a response file

To install:

1. Log on as the user who installed the WebSphere Application Server product.

— 

AIX	HP-UX	Linux	Solaris
-----	-------	-------	---------

Log on using the same user ID that was used when the product was installed.

Run the `ls -al` command at the root of the application server installation to find the user ID that was used to install the product.

— 

Windows
---------

Log on as a user who belongs to the administrator group or as the user who installed the product.

2. Extract the downloaded product files. Save a back-up copy of the response file `responsefile.WEB2FEP.txt`.
3. Edit the `responsefile.WEB2FEP.txt` file and customize it for the installation environment.

Set values for the options shown in Table 4-1 and save the file.

Table 4-1 Silent install response file options on distributed operating systems

Option	Description
-OPT silentInstallLicenseAcceptance="true"	The enclosed license agreement to install Feature Pack for Web 2.0 must be accepted.
-OPT installLocation=<app_server_root>	Set <app_server_root> to the root directory of the application server product.
-OPT allowNonRootSilentInstall="true"	IBM WebSphere Application Server Version 6.1 only. Uncomment this line if installing the product with a non-root or non-administrator ID.

To override any option in the response file with command-line options, include the following option in the -OPT overrideResponsefileOptions=true response file.

- Stop the application server and any other product processes for the application server on which Feature Pack for Web 2.0 is being installed. For example, for the Network Deployment product, stop all application server processes, the node agent, and the deployment manager.
- Run the Feature Pack for Web 2.0 installation program using the -silent option.

Run the install executable file in the root directory of the feature pack installation image and point to the Java Virtual Machine (JVM) already included with the application server, where <app\_server\_root> is the root directory of the existing application server product. At a command prompt in the root directory of the installation image, enter the following command:

```

- AIX HP-UX Linux Solaris
./install -is:javahome <app_server_root>/java -options
responsefile.WEB2FEP.txt -silent

- Windows
install -is:javahome <app_server_root>\java -options
responsefile.WEB2FEP.txt -silent

```

The installation program runs silently in the background.

The installation wizard will then install Feature Pack for Web 2.0 and record the installation events in the installation log files in the <app\_server\_root>/logs/install/Web2fep directory.

### 4.5.3 Installing Feature Pack for Web 2.0 on i5/OS

Install Feature Pack for Web 2.0 directly from the iSeries® server onto an existing IBM WebSphere Application Server Version 6.x.

Before starting this installation, complete the following actions:

1. Ensure that the user profile has \*ALLOBJ and \*SECADM special authorities.
2. Determine whether the application server is already installed on the server. The Feature Pack for Web 2.0 must be installed on an existing application server installation.
  - a. Enter the Display Software Resources (DSPSFWRSC) command on a CL command line.
  - b. Look for an entry with the product resource ID:
    - IBM WebSphere Application Server Version 6.0: 5733W60
    - IBM WebSphere Application Server Version 6.1: 5733W61

If the product resource ID is not found, then this product has not been installed on the iSeries server.

**Note:** If the product resource ID is found, ensure that at least one of the following entries is displayed:

- ▶ IBM WebSphere Application Server Version 6.0
  - 5733W60 2 5102 WebSphere Application Server V6.0
  - 5733W60 1 5101 WebSphere Application Server V6.0 Express
  - 5733W60 3 5103 WebSphere Application Server V6.0 Network Deployment
- ▶ IBM WebSphere Application Server Version 6.1
  - 5733W61 2 5102 WebSphere Application Server V6.1
  - 5733W61 1 5101 WebSphere Application Server V6.1 Express
  - 5733W61 3 5103 WebSphere Application Server V6.1 Network Deployment

3. Install IBM WebSphere Application Server Version 6.x if you have not already done so. Update the application server to the required version as specified in 4.3.1, “Updating the application server” on page 45.

4. Download the feature pack (see 4.4, “Downloading Feature Pack for Web 2.0” on page 45) and extract the contents to an integrated file system (IFS) directory on the iSeries system on which Feature Pack for Web 2.0 will be installed. For example, create two directories, *Web20\_download* and *Web20\_installation\_image*. Download the feature pack to the *Web20\_download* directory and extract the downloaded file to the *Web20\_installation\_image* directory. Do not move any files out of the *Web20\_installation\_image* directory.

### Installing silently using the installation program

Feature Pack for Web 2.0 can be installed from Qshell using the `INSTALL` command. The installation wizard can also be invoked by using the `RUNJVA` command.

To install:

1. Log on to the iSeries system with a user profile that has `*ALLOBJ` and `*SECADM` special authorities.
2. Copy the `responsefile.WEB2FEP.txt` file located in the root directory of the extracted install image.
3. Edit the body of the response file. The file can be edited using a mapped drive and a text editor or by using the `EDTF CL` command from the i5/OS system command line.

Set values for the options shown in Table 4-2 and save the file.

Table 4-2 Silent install response file options on i5/OS

Option	Description
<code>-OPT silentInstallLicenseAcceptance="true"</code>	The enclosed license agreement to install Feature Pack for Web 2.0 must be accepted.
<code>-OPT installLocation=&lt;app_server_root&gt;</code>	Set <code>&lt;app_server_root&gt;</code> to the root directory of the application server product.

To override any option in the response file with command-line options, include the following option in the response file:

`-OPT overrideResponsefileOptions=true`

4. Stop the application server and any other product processes for the application server on which Feature Pack for Web 2.0 is being installed. For example, for the Network Deployment product, stop all application server processes, the node agent, and the deployment manager.

5. Verify that the host server jobs have started on the iSeries server.

The host server jobs allow the installation code to run on iSeries. On a CL command line, enter the following command:

```
STRHOSTSVR SERVER(*ALL)
```

6. Read the IBM International Program License Agreement located in the lafiles directory.

If agreed to the terms of the agreement, continue with the installation process.

7. Invoke the installation program for the feature pack.

Run the INSTALL command from Qshell or use the RUNJVA command from the CL command line.

In the following example commands, *path/responsefile* represents the fully qualified path of the response file edited:

- Run the INSTALL command from Qshell.

- i. On a CL command line, issue the STRQSH command to start the Qshell command shell.
- ii. Issue the installation command from the root directory of the extracted install image to start the installation program:

```
install -options path/responsefile
```

**Important:** Do not exit the Qshell session (PF3) until the installation has completed. Doing so might cause the installation to stop prematurely.

- Or issue the RUNJVA command from the CL command line.

At the CL command line, enter the RUNJVA command in the root directory of the extracted install image. Enter the full command on one line. The command is shown here on more lines for formatting clarity:

```
RUNJVA
CLASS(run) PARM('-options' 'path/responsefile')
CLASSPATH('setup.jar')
PROP(
  ('Xbootclasspath/p' 'java/endorsed/xml.jar')
  (java.version 1.5)
  (is.debug 1)
)
```

The installation wizard will then install Feature Pack for Web 2.0 and record the installation events in the installation log files in the <app\_server\_root>/logs/install/Web2fep directory.



8. For security purposes, if the host servers were not running prior to installation, we recommend that the end host server (ENDHOSTSVR) command is run after the installation is complete.

#### **4.5.4 Installing Feature Pack for Web 2.0 on z/OS**

The various features of the feature pack have been made available as a simple zip file without platform requirements to be compatible with any development environment (see 4.4, “Downloading Feature Pack for Web 2.0” on page 45).

The contents of this feature pack are targeted towards application development environments and the tooling of these development efforts. However, enough free disk space must exist to install the feature pack. The requirement is 150 MB of free space available in the target installation directory.

To download and extract the Feature Pack for Web 2.0:

1. Create two directories, Web20\_download and Web20\_extracted.
2. Download the feature pack (see 4.4, “Downloading Feature Pack for Web 2.0” on page 45) to the Web20\_download directory and then extract the downloaded zip file to the Web20\_extracted directory. This extracted directory contains the various features under named directories, and the individual components of the features can be imported as needed.

After the Feature Pack for Web 2.0 downloaded archive has been extracted it can be used by working with the various included features and applications.

### **4.6 Uninstalling Feature Pack for Web 2.0**

The following sections outline the installation instructions for uninstalling Feature Pack for Web 2.0 for the appropriate operating system.

#### **4.6.1 Uninstalling Feature Pack for Web 2.0**

The Feature Pack for Web 2.0 uninstaller removes the feature pack product files, leaving the application server product intact.

The uninstaller program is customized for each product installation, with specific disk locations and routines for removing installed features.

Any feature packs should be uninstalled first before the application server is uninstalled. If a feature pack is installed, uninstalling the WebSphere Application Server product using the uninstaller program causes the feature pack to fail.

However, the feature pack should still be uninstalled after uninstalling the application server to remove all feature pack product entries and artifacts that might prevent a successful reinstallation. See 4.6.4, “Troubleshooting” on page 60, for more information. This same limitation applies if a customized installation package (CIP) created with the Installation Factory is being uninstalled.

1. Log on as the user who installed the application server product.

– **AIX**    **HP-UX**    **Linux**    **Solaris**

Log on using the same user ID that was used when the product was installed.

Run the `ls -al` command at the root of the application server installation to find the user ID that was used to install the product.

– **Windows**

Log on as a user who belongs to the administrator group or as the user who installed the product.

2. If the feature pack is installed on a Network Deployment product, then stop the node agent process with the **stopNode** command.

Stop the node agent process that might be running on the machine. For example, issue the following command from the `<profile_root>/bin` directory of a federated node on a Linux workstation to stop the node agent process:

```
./stopNode.sh
```

If servers are running and security is enabled, then use the following command:

```
./stopNode.sh -user user_ID -password password
```

3. If the feature pack is installed on a Network Deployment product, then stop the deployment manager dmgr process with the **stopManager** command. For example, issue the following command on a Linux workstation from the `<profile_root>/bin` directory of the deployment manager profile:

```
./stopManager.sh -user user_ID -password password
```

4. Stop each running application server with the **stopServer** command.

Stop all server processes in all profiles on the machine. For example, issue the following command from the `<profile_root>/bin` directory to stop the `server1` process in the application server profile:

```
./stopServer.sh server1
```

If a server is running and security is enabled, use the following command:

```
./stopServer.sh server1 -user user_ID -password password
```

For multiple servers, use the **serverStatus** command to find running application servers. Issue the following command from the *<profile\_root>/bin* directory to determine which servers, if any, are running:

```
./serverStatus.sh -all
```

- Optional: Back up configuration files, profiles, and log files to refer to them later if necessary.

Use the **backupConfig** command to back up configuration files and profiles.

- Uninstall Feature Pack for Web 2.0 using the uninstallation wizard or a silent uninstallation.

## Uninstalling using the uninstallation wizard graphical interface

To uninstall run the uninstall command in the *<app\_server\_root>/uninstall\_Web2fep* directory:

- AIX** | **HP-UX** | **Linux** | **Solaris**

```
uninstall.sh
```

- Windows**

```
uninstall.exe
```

The uninstallation wizard begins and displays the Welcome panel. Click **Next** to begin uninstalling the product.

The uninstaller wizard displays a confirmation panel that lists a summary of the components to be uninstalled.

- Click **Next** to continue uninstalling the product.
- Click **Finish** to close the wizard after the wizard removes the product.

## Uninstalling silently

To uninstall:

- To uninstall silently, run the uninstall command shown in Table 4-3.

Table 4-3 Silent uninstallation option on distributed operating systems

Option	Description
uninstall -silent	Use this command to uninstall the feature pack silently on the base or express editions of WebSphere Application Server.

- Review the uninstallation log files located in the *<app\_server\_root>/logs/uninstall/Web2fep* directory.
- If necessary, delete the remaining */uninstall\_Web2fep* directory.

The Feature Pack for Web 2.0 is now uninstalled.

See 4.5.2, “Installing Feature Pack for Web 2.0 on distributed OSs” on page 47, to reinstall Feature Pack for Web 2.0.

## 4.6.2 Uninstalling the feature pack on i5/OS

Feature Pack for Web 2.0 can be uninstalled by running the feature pack uninstall command from an iSeries server. The uninstall command calls the uninstaller program that is created during installation. The uninstaller program is customized for each product installation, with specific disk locations and routines for removing installed features.

Any feature packs should be uninstalled before the application server is uninstalled. If a feature pack is installed, uninstalling the WebSphere Application Server product using the uninstaller program causes the feature pack to fail. However, the feature pack should still be uninstalled after uninstalling the application server to remove all feature pack product entries and artifacts that might prevent a successful reinstallation. See 4.6.4, “Troubleshooting” on page 60, for more information. This same limitation applies if a customized installation package (CIP) created with the installation factory is being uninstalled.

1. Log on to the iSeries system with a user profile that has \*ALLOBJ special authority.
2. Stop all processes for the application server product for which Feature Pack for Web 2.0 is being uninstalled.

Stop the application server and any other product processes. For example, for the Network Deployment product, stop the deployment manager, the node agent, and all application server processes.

3. Verify that the host server jobs have started on the iSeries server.

The host server jobs allow the installation code to run on iSeries. On a CL command line, enter the following command:

```
STRHOSTSVR SERVER(*ALL)
```

4. Uninstall Feature Pack for Web 2.0 from the WebSphere Application Server for i5/OS installation on the iSeries server.

To uninstall the feature pack, run the command shown in Table 4-4 from the Qshell command line in the <app\_server\_root>/uninstall\_Web2fep directory.

Table 4-4 Silent uninstallation options on i5/OS

Option	Description
uninstall -silent	Use this command to uninstall the feature pack silently on the base or express editions of WebSphere Application Server.

Uninstalling Feature Pack for Web 2.0 from the iSeries server removes the feature pack libraries and directories.

5. Review any log files if necessary. Log files are located in the <app\_server\_root>/logs/uninstall/Web2fep directory.

After completing the procedure, the feature pack is uninstalled.

If the feature pack does not uninstall successfully, examine the logs in the app\_server\_root/logs/uninstall/Web2fep directory to identify why the uninstallation failed. Correct the problems identified and try uninstalling the product again.

### 4.6.3 Manual uninstallation

If an error was encountered during installation or uninstallation, then follow the instructions in this section to manually remove the feature pack.

#### IBM WebSphere Application Server Version 6.0

For this:

1. Delete the following directories:
  - <app\_server\_root>/Web2fep
  - <app\_server\_root>/lfiles/Web2fep
  - <app\_server\_root>/properties/version/nif/Web2fep
  - <app\_server\_root>/uninstall\_Web2fep
2. Delete the <app\_server\_root>/properties/WEB2FEPCustomConstants.properties file.
3. Delete the following files from <app\_server\_root>/properties/version:
  - ajax.client.runtime.component
  - ajax.proxy.component
  - dev.guide.componeny
  - eclipse.site.component

- feedsphere.component
  - legal.Web2fep.component
  - lib.json4j.component
  - productspecificfiles.Web2fep.component
  - rpc.adapter.component
  - samples.feedsphere.component
  - samples.pbw.60.component
  - samples.quotestreamer.component
  - samples.rpc.component
  - WEB2FEP.product
4. Delete the following files from the <app\_server\_root>/properties/version/update/backup directory:
- fep.Web2fep.primary.pak
  - fep.Web2fep.common.pak
  - fep.Web2fep.60.pak
5. Remove the following directories from <app\_server\_root>/properties/version/install/6.0.0.0/backup:
- ajax.client.runtime
  - ajax.proxy
  - dev.guide
  - eclipse.site
  - feedsphere
  - legal.Web2fep
  - lib.json4j
  - nif.componentmap.Web2fep.60
  - nif.componentmap.Web2fep.common
  - nif.componentmap.Web2fep.primary
  - productspecificfiles.Web2fep
  - rpc.adapter
  - samples.feedsphere
  - samples.pbw.60
  - samples.quotestreamer
  - samples.rpc
6. Delete the following files from <app\_server\_root>/properties/version/install/6.0.0.0/backup/componentmaps:
- componentmap.Web2fep.60.xml
  - componentmap.Web2fep.common.xml
  - componentmap.Web2fep.primary.xml
7. Download and install the latest version of the Update Installer. The included **installRegistryUtils** command is needed to manually remove the feature pack entries from the product registry.

8. Remove the WEB2FEP entries from the product registry with the **installRegistryUtils** command in the `<updi_root>/bin` directory, where `<updi_root>` is where the Update Installer is installed, typically in `<app_server_root>/UpdateInstaller`.

```
./installRegistryUtils.bat|sh -cleanProduct -offering WEB2FEP  
-installLocation <app_server_root>
```

## IBM WebSphere Application Server Version 6.1

For this:

1. Delete the following directories:
  - `<app_server_root>/Web2fep`
  - `<app_server_root>/lafiles/Web2fep`
  - `<app_server_root>/properties/version/nif/Web2fep`
  - `<app_server_root>/uninstall_Web2fep`
2. Delete the `<app_server_root>/properties/WEB2FEPCustomConstants.properties` file.
3. Delete the following files from `<app_server_root>/properties/version`:
  - `ajax.client.runtime.component`
  - `ajax.proxy.component`
  - `dev.guide.component`
  - `eclipse.site.component`
  - `feedsphere.component`
  - `legal.Web2fep.component`
  - `lib.json4j.component`
  - `productspecificfiles.Web2fep.component`
  - `rpc.adapter.component`
  - `samples.feedsphere.component`
  - `samples.pbw.61.component`
  - `samples.quotestreamer.component`
  - `samples.rpc.component`
  - `WEB2FEP.product`
4. Delete the following files from the `<app_server_root>/properties/version/nif/backup` directory:
  - `fep.Web2fep.primary.pak`
  - `fep.Web2fep.common.pak`
  - `fep.Web2fep.61.pak`
5. Remove the following directories from `<app_server_root>/properties/version/nif/backup`:
  - `ajax.client.runtime`
  - `ajax.proxy`

- dev.guide
  - eclipse.site
  - feedsphere
  - legal.Web2fep
  - lib.json4j
  - nif.componentmap.Web2fep.61
  - nif.componentmap.Web2fep.common
  - nif.componentmap.Web2fep.primary
  - productspecificfiles.Web2fep
  - rpc.adapter
  - samples.feedsphere
  - samples.pbw.61
  - samples.quotestreamer
  - samples.rpc
6. Delete the following files from  
<app\_server\_root>/properties/version/install/nif/componentmaps:
- componentmap.Web2fep.61.xml
  - componentmap.Web2fep.common.xml
  - componentmap.Web2fep.primary.xml
7. Remove the WEB2FEP entries from the product registry with the **installRegistryUtils** command in the <app\_server\_root>/bin directory:
- For a root user or administrator:  

```
./installRegistryUtils.bat|sh -cleanProduct -offering WEB2FEP
-installLocation <app_server_root>
```
  - For a non-root or non-administrator user, where <user\_home> is the home directory of the current user:  

```
./installRegistryUtils.bat|sh -cleanProduct -offering WEB2FEP
-installLocation <app_server_root> -userHome <user_home>
```

## 4.6.4 Troubleshooting

### Linux

Uninstalling the application server before uninstalling the feature pack might result in the failure of configuration actions that are responsible for removing Linux shortcuts. Uninstall any feature pack installations first before uninstalling the application server to ensure that both products are uninstalled cleanly. If the application server has already been uninstalled, then run the feature pack uninstaller to clean up the feature pack installation and to remove any product registry entries. Ignore the failed configuration actions from the application server



uninstallation process. However, manually remove the entire application server directory afterwards.

Archived



# Application development tools

This chapter presents a description of the tooling helpful in Ajax application development, most of which we utilized in the preparation of the scenarios described in later chapters.

In this chapter we provide information about each tool, the specific installation steps required, and their development environment setup and usage.

The sections in this chapter are:

- ▶ Ajax development tools overview
- ▶ Rational Application Developer 7.0.0.6
- ▶ Eclipse 3.3 with WAS CE 2.0 WTP Server Adapter
- ▶ Firefox and Firebug
- ▶ Internet Explorer debugging aids
- ▶ JSON Formatter

## 5.1 Ajax development tools overview

Ajax technology introduces several new protocols. To develop, debug, and test complex Web applications productively we need efficient development tools. Successful deployment of Ajax applications requires development tools that can hide much of the complexity.

This section introduces the development tools that we found helpful on the server side as well as the browser side development of the Web 2.0 applications used in the scenarios presented.

## 5.2 Rational Application Developer 7.0.0.6

The IBM Rational Application Developer for WebSphere Software application helps Java developers create Java/Java Platform, Enterprise Edition (Java EE), portal, Web, Web services, and service-oriented architecture (SOA) applications. This integrated development environment (IDE) assists in rapidly designing, developing, assembling, testing, and deploying these applications. Its visual tools can be leveraged to reduce manual coding by abstracting the Java EE programming model, making it easier for those less familiar with Java technology to complete development projects. It is designed to be a highly flexible development tool based on the Eclipse open framework and with a range of flexible installation options.

### 5.2.1 IBM Installation Manager

IBM Installation Manager is used to install Rational Application Developer. IBM Installation Manager is a program that helps you install the Rational desktop product packages on your workstation. It also helps you update, modify, and uninstall other packages. A package can be a product, a group of components, or a single component that is designed to be installed by Installation Manager.

There are six wizards in the Installation Manager that make it easy to maintain your package life cycle, as shown in Figure 5-1. These six wizards are:

- ▶ The Install Packages wizard walks you through the installation process.
- ▶ The Update Packages wizard searches for available updates to installed packages.
- ▶ The Modify Packages wizard is helpful in modifying certain elements of installed packages.
- ▶ The Manage Licenses wizard helps you set up the licenses for your installed packages.
- ▶ The Roll Back Packages wizard can help you revert back to a previous version of a package.
- ▶ The Uninstall Packages wizard removes an installed package from your system.



Figure 5-1 IBM Installation Manager

## 5.2.2 Installation of Rational Application Developer

There are a number of scenarios that you can follow when installing Rational Application Developer:

- ▶ Installing from the CDs
- ▶ Installing from a downloaded electronic image on your workstation
- ▶ Installing from an electronic image on a shared drive
- ▶ Installing from a repository on an HTTP or HTTPS server

Note that in the latter three scenarios, you can choose to run the installation manager program in silent mode to install Rational Application Developer. See the installation guide for details on each scenario. While writing this book we installed IBM Rational Application Developer V7.0 from a downloaded electronic image on the workstation. The steps are:

1. Start the Rational Application Developer Installer by running launchpad.exe from the disk1 folder.
2. The IBM Rational Application Developer V7.0 components have separate installations from the main Launchpad Base page, as seen in Figure 5-2.



Figure 5-2 Launchpad of Rational Application Developer

3. Select **Install IBM Rational Application Developer v7.0**. You are directed to install IBM Installation Manager because it is not installed yet, as seen in Figure 5-3. Click **Next**.

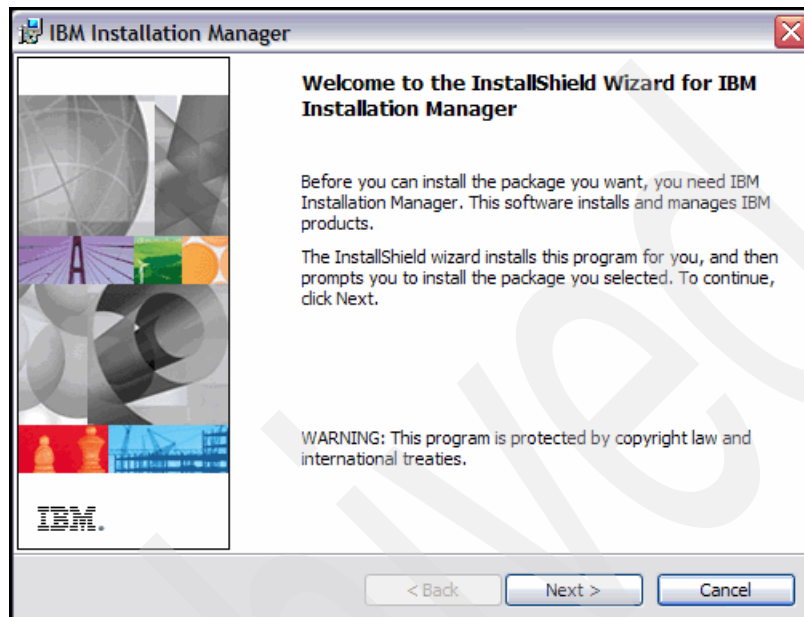


Figure 5-3 Welcome screen of Installation Manager installation

4. In the License Agreement window, review the terms in the license agreement, select **I accept the terms of the license agreement**, and click **Next**.
5. You are directed to the destination folder of IBM Installation Manager. Click **Next**, as shown in Figure 5-4.

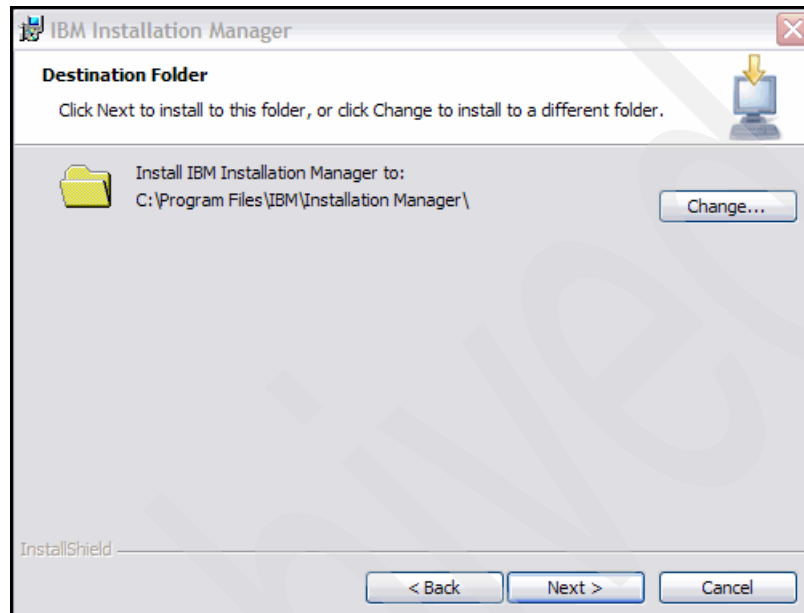


Figure 5-4 Destination folder for Installation Manager

6. On the Install Packages page you will find that IBM Rational Application Developer 7.0.0. is already selected. Click **Next**, as shown in Figure 5-5.

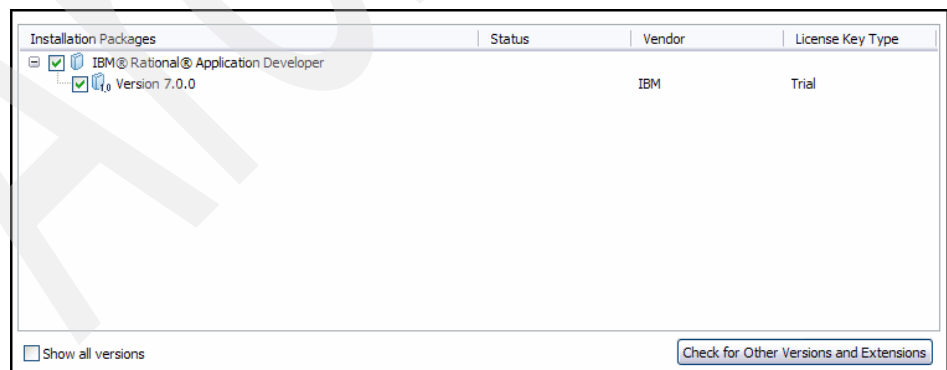


Figure 5-5 Rational Application Developer 7.0.0 Installation selection box

7. Select **I accept the terms in the license agreements** and then click **Next**.



8. On the Select a location for the shared resources directory page you can change the shared resource directory location. Click **Next**, as shown in Figure 5-6.

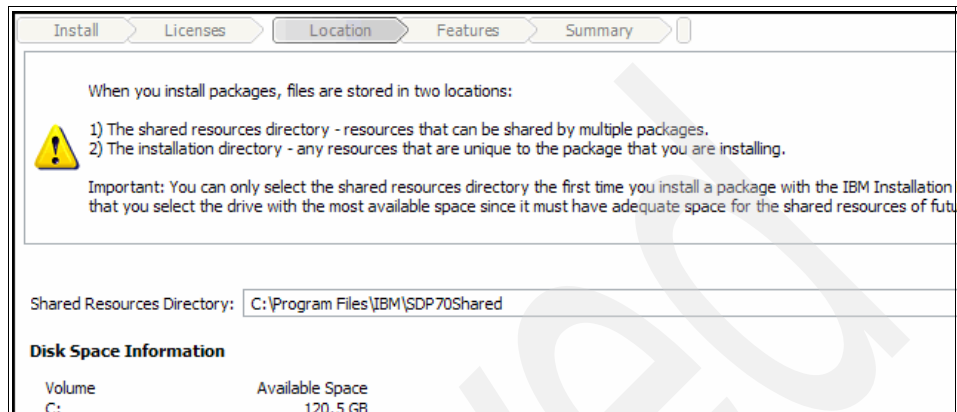


Figure 5-6 Shared resources directory

9. On the Package Group page, select **Create a new package group**, which allows you to change the installation directory for the package group. Click **Next**, as shown in Figure 5-7.

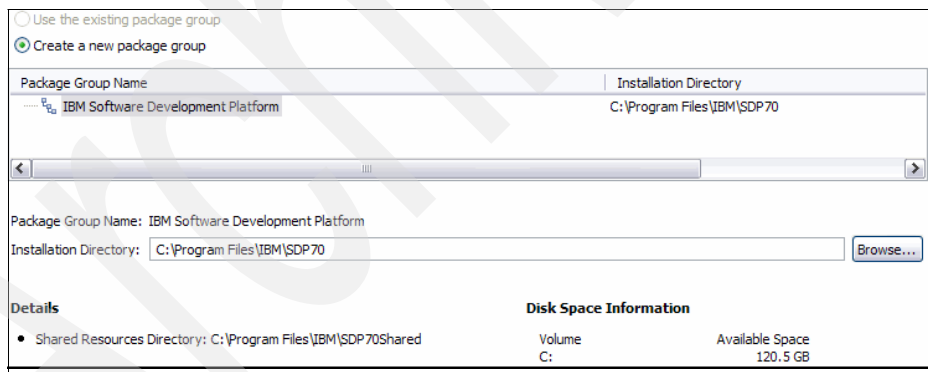


Figure 5-7 Installation directory

10. IBM Rational Application Developer provides you with the option of extending the existing Eclipse version. If you have a compatible version of Eclipse already installed you can extend it functionally by specifying the path for the Eclipse IDE and JVM.

For this book's purposes we continued by using the default installation. Click **Next**, as shown in Figure 5-8.

The screenshot shows the 'Install Packages' dialog box with the 'Location' tab selected. The dialog has a title bar and a navigation bar with tabs: 'Install', 'Licenses', 'Location', 'Features', and 'Summary'. Below the navigation bar, there is a warning icon and text: 'Packages are bundled with a version of the Eclipse integrated development environment (IDE or workbench). Click Next for installation. However, if you already have a compatible version of the Eclipse IDE on your system, you can extend that version of Eclipse with a new one. When you extend Eclipse, the package functions are available in your Eclipse IDE, but the package files are stored in a separate directory. For more information, see the installation guide for a package that you are installing.' Below this text, there is a checkbox labeled 'Extend an existing Eclipse'. Underneath the checkbox, there are two text input fields: 'Eclipse IDE:' and 'Eclipse JVM:'.

Figure 5-8 Extend an existing Eclipse IDE

11. Choose the languages that you wish to support. Click **Next**, as shown in Figure 5-9.

The screenshot shows the 'Install Packages' dialog box with the 'Features' tab selected. The dialog has a title bar and a navigation bar with tabs: 'Install', 'Licenses', 'Location', 'Features', and 'Summary'. Below the navigation bar, there is a section titled 'Languages' with a list of checkboxes for different languages. The 'English' checkbox is checked. The other languages are: Brazilian Portuguese, Czech, French, Hungarian, Italian, Japanese, Korean, Russian, Simplified Chinese, Spanish, and Traditional Chinese.

Figure 5-9 Selection of language

12. Select the package features to install. You can select all features with WebSphere Application Server Version 6.1. We are installing WebSphere Application Server test environment with this installation of Application Developer for application testing and debugging purposes. Click **Next**, as shown in Figure 5-10.

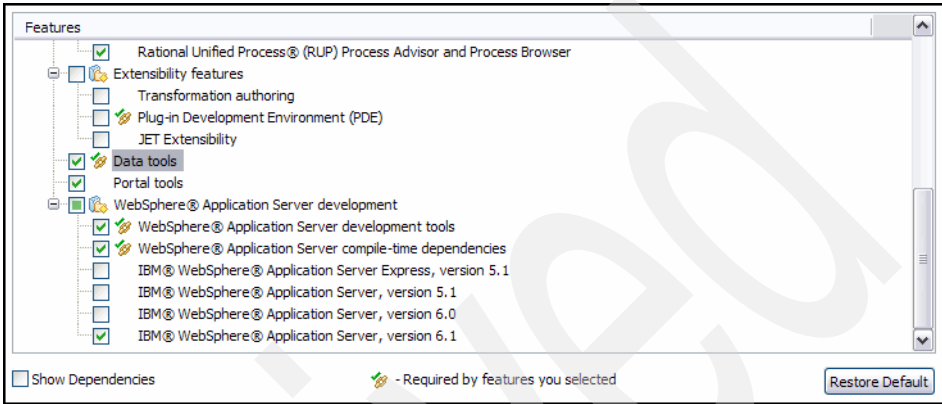


Figure 5-10 Selection of packages

13. You are now presented with the Summary page. Review all features selected for installation and then click **Install**, as shown in Figure 5-11.

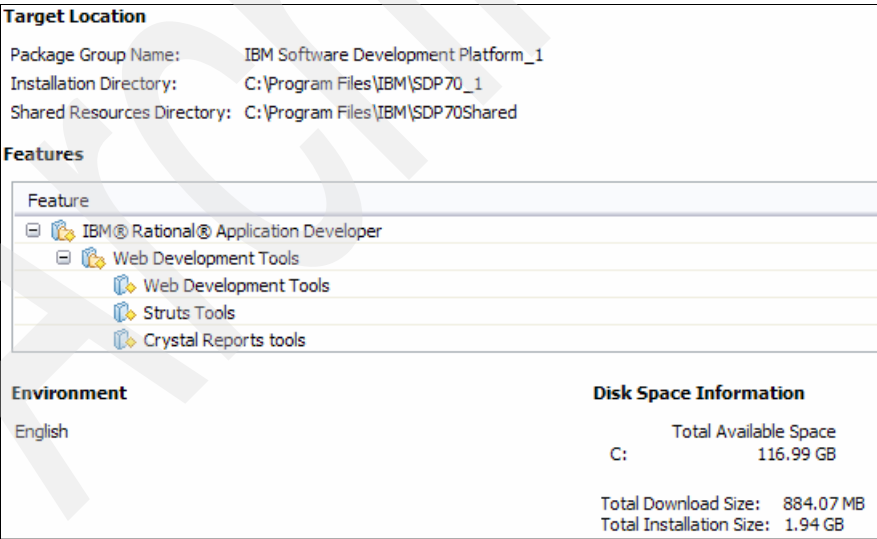


Figure 5-11 Summary of packages being installed

14. When the installation is complete you are presented with the confirmation page, as shown in Figure 5-12. You can now launch the Rational Application Developer IDE.

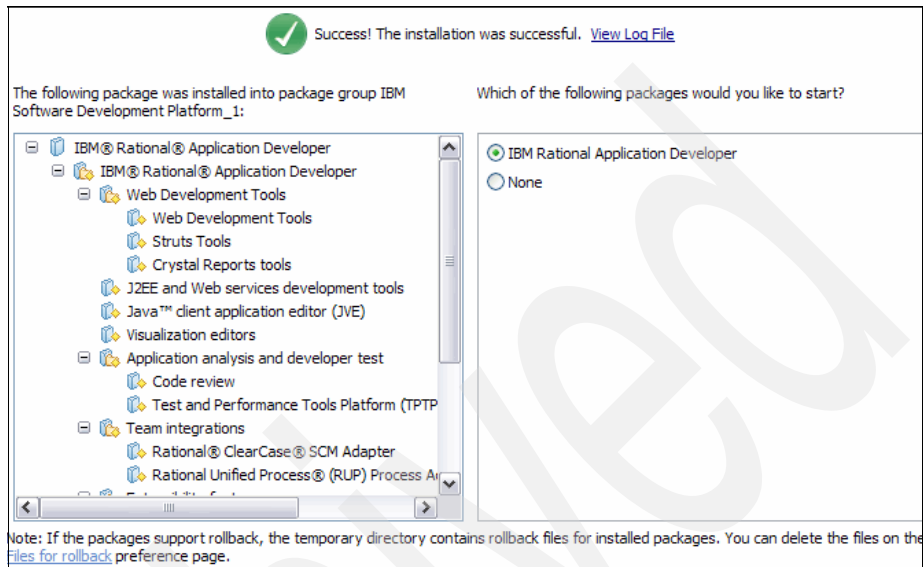


Figure 5-12 Installation complete

### 5.2.3 Upgrading IBM Installation Manager

IBM continuously releases updates to the IBM Installation Manager software, which can be downloaded from the IBM support Web site. At the time of writing, the latest version of the IBM Installation Manager available is v1.1.1, and we upgraded to this version. All supported files can be downloaded from the following Web site:

[http://www-1.ibm.com/support/docview.wss?rs=2044&context=SSCM72&dc=D400&uid=swg24018338&loc=en\\_US&cs=UTF-8&lang=en&rss=ct2044rational](http://www-1.ibm.com/support/docview.wss?rs=2044&context=SSCM72&dc=D400&uid=swg24018338&loc=en_US&cs=UTF-8&lang=en&rss=ct2044rational)

If your system is connected to the Internet you can upgrade the Installation Manager online, in which case you can omit steps 1–5.

1. Click **OK**.
2. Click **OK** on the preferences panel to close it and return to the Installation Manager panel.

3. Once you are back to the Installation Manager main page (Figure 5-1 on page 65) double-click the Update Packages icon. A window pops up, as shown in Figure 5-13, indicating that a new version of the IBM Installation Manager is available.
4. Select **Yes** to upgrade.

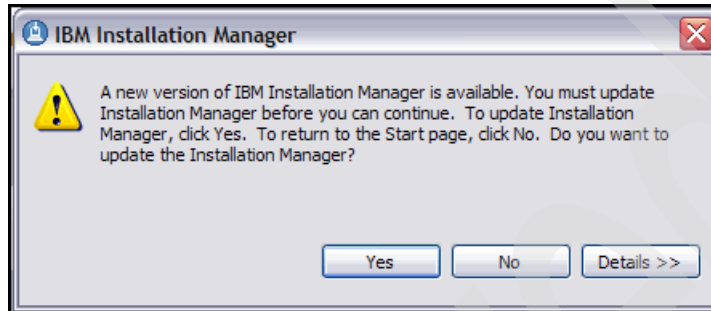


Figure 5-13 Update available for Installation Manager

5. When the update is complete, restart the IBM Installation Manager, as shown in Figure 5-14.

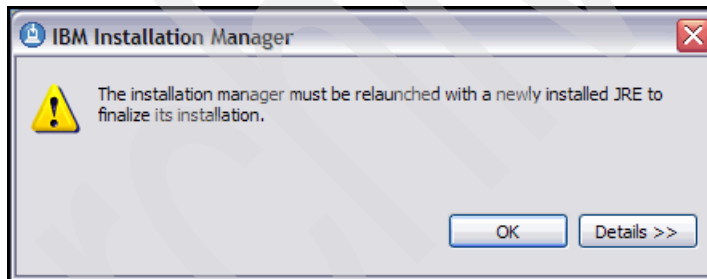


Figure 5-14 Upgrade of IBM Installation Manager complete

## 5.2.4 Upgrading IBM Rational Application Developer

IBM releases update fixpacks of Rational Application Developer approximately every 3 months. Fixpacks can be downloaded from IBM support Weblink. During the preparation of this book the latest version available for IBM Rational Application Developer was 7.0.0.6 and we upgraded to this version.

Fixpacks are available for download from the following Web site:

[http://www-1.ibm.com/support/docview.wss?rs=2042&context=SSRTLW&dc=D400&uid=swg24018313&loc=en\\_US&cs=UTF-8&lang=en&rss=ct2042rational](http://www-1.ibm.com/support/docview.wss?rs=2042&context=SSRTLW&dc=D400&uid=swg24018313&loc=en_US&cs=UTF-8&lang=en&rss=ct2042rational)

To upgrade our Rational Application Developer 7.0.0.6 utilizing downloaded files:

1. Download the rad7006.update.disk1.zip file and extract it to a local directory on your workstation.
2. Open the IBM Installation Manager and click **File** → **Preferences**.
3. Click **Add Repository** and browse for the diskTag.inf file in the folder in which you extracted the downloaded rad7006.update.disk1.zip file, as shown in Figure 5-15.

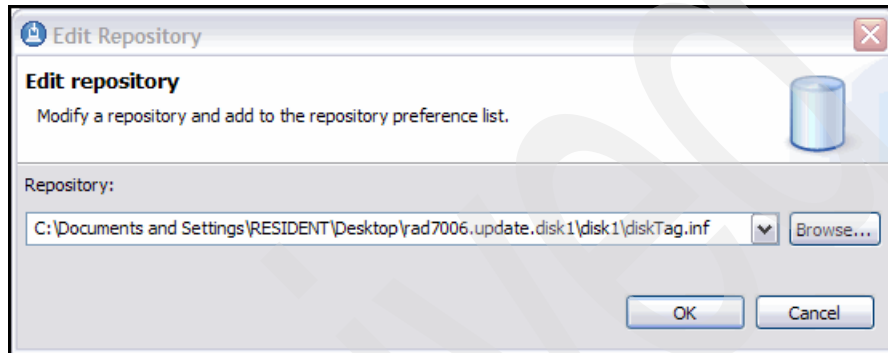


Figure 5-15 Add Rational Application Developer update repository

4. Click **OK** and the full path name appears in the Edit repository panel of the Installation Manager.
5. Click **OK** on the Preferences panel to close it and return to the Installation Manager panel.
6. On the Installation Manager panel, double-click the Update Packages icon.

7. Select **IBM Software Development Platform** and click **Next**, as shown in Figure 5-16.

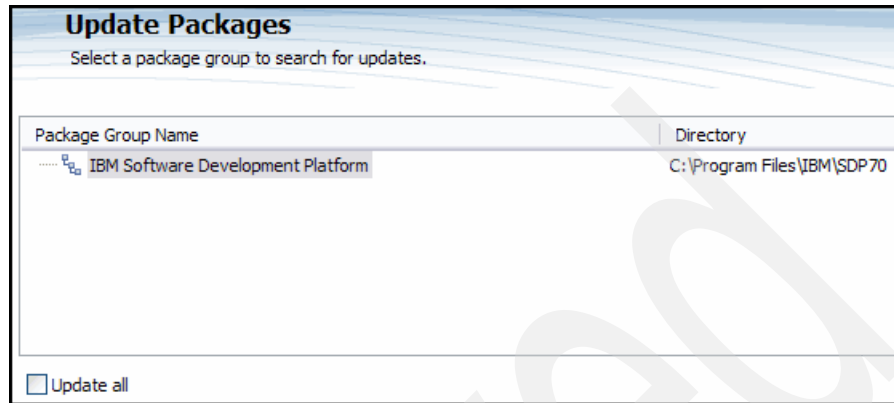


Figure 5-16 Select products to update

8. Select **IBM Rational Application Developer Version 7.0.0.6** and click **Next**, as shown in Figure 5-17.

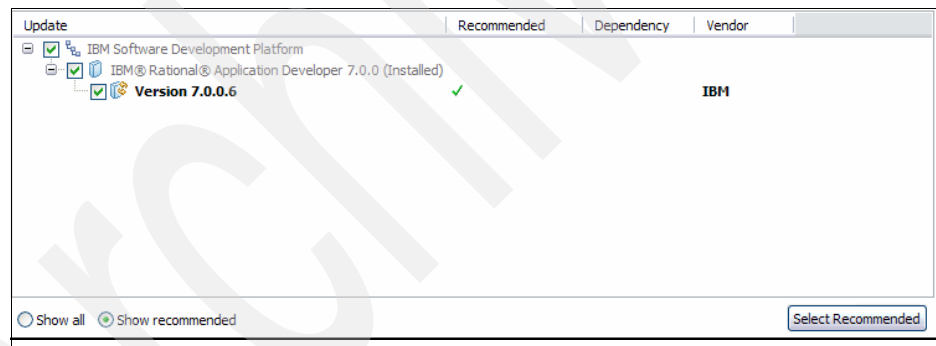



Figure 5-17 Update version available

9. After reading and accepting the license agreement click **Next**.

10.) Review the update packages to be installed and then click **Update**, as shown in Figure 5-18.

**Target Location**  
Package Group Name: IBM Software Development Platform  
Installation Directory: C:\Program Files\IBM\SDP70  
Shared Resources Directory: C:\Program Files\IBM\SDP70Shared

**Updates**

Update	Package	Package Group Name
 Version 7.0.0.6	IBM® Rational® Application Dev...	IBM Software Development Platform

**Disk Space Information**  

Total Available Space  
C: 115.84 GB

Total Download Size: 978.67 MB  
Total Installation Size: 1.21 GB

**Repository Information**  
Files will be retrieved from the following locations:

Repository	Download Size
C:\Rad7.0\disk1	29.8 KB
C:\Documents and Settings\RESIDENT\Desktop\rad7006.update.disk1\disk1	978.64 MB

Figure 5-18 Update packages to install

11. We may need other update files like rad7006.update.disk2.zip, rad7006.update.disk3.zip, rad7006.update.disk4.zip, and so on. You can download and extract these files as mentioned in step 1 for rad7006.update.disk1.zip.



12. Refer to the location of disk 2, disk 3, and so on, when prompted, as shown in Figure 5-19.

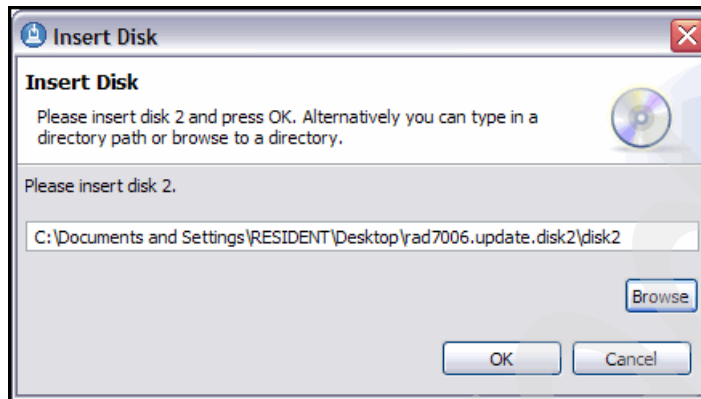


Figure 5-19 Browse for disk 2

13. Once the update is finished you can review the log file and click **Finish**, as shown in Figure 5-20.

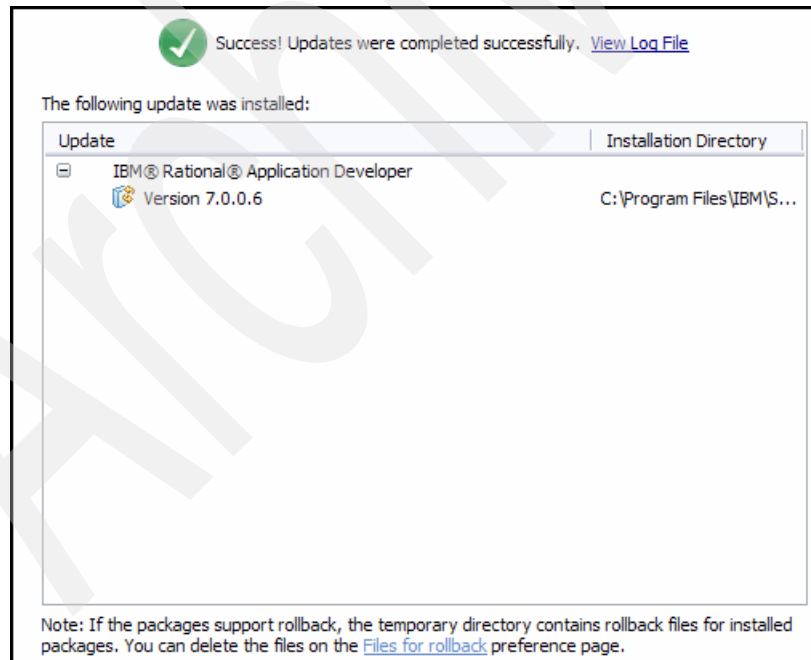


Figure 5-20 Update complete

## 5.2.5 Configuring Web 2.0 feature pack Eclipse plug-in

If you have installed WebSphere Application Server Feature Pack for Web 2.0 (as instructed in Chapter 4, “Installation” on page 43) and Rational Application Developer, we can configure Eclipse plug-in for Web 2.0 feature pack. To configure the Web 2.0 Eclipse plug-in with Rational Application Developer:

1. In the Rational Application Developer workbench go to **Help** → **Software Updates** → **Find and Install**.
2. Select **Search for new features to install** in the pop-up window and click **Next**.
3. Click **New Archived Site** and browse for eclipse-update-site.zip in <<installation directory of WAS>>/Web2fep/site. Click **OK**, as shown in Figure 5-21.

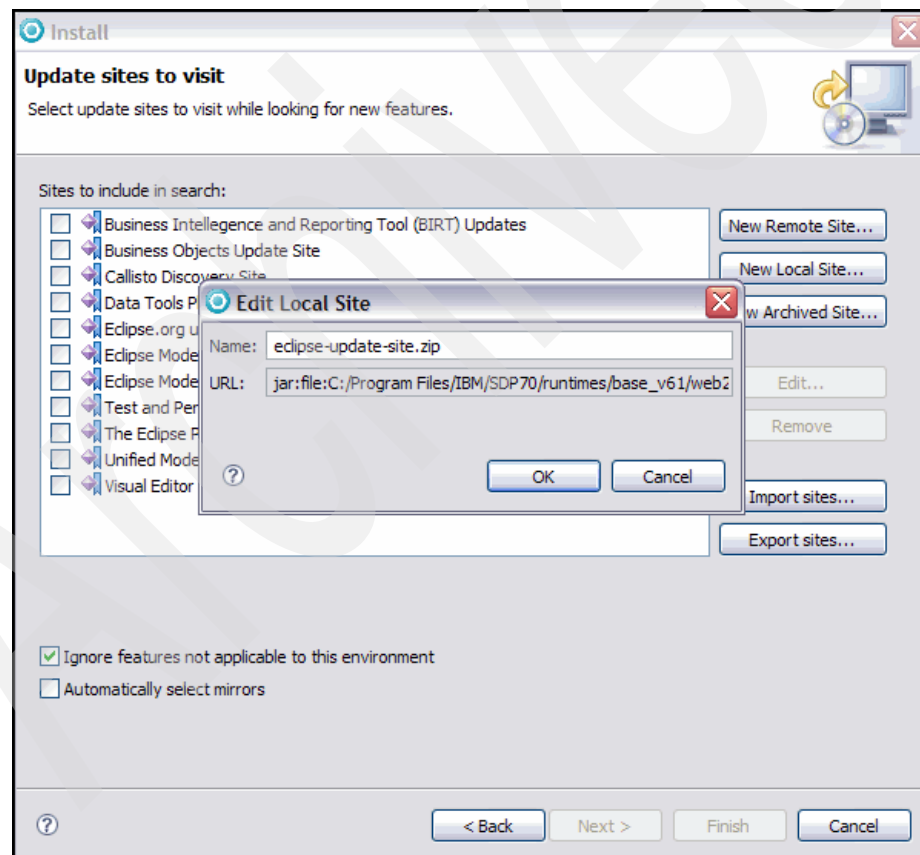


Figure 5-21 Add update site

- Click **Finish**. The pop-up panel displays a list of features to be installed. Select all the features and click **Next**, as shown in Figure 5-22.

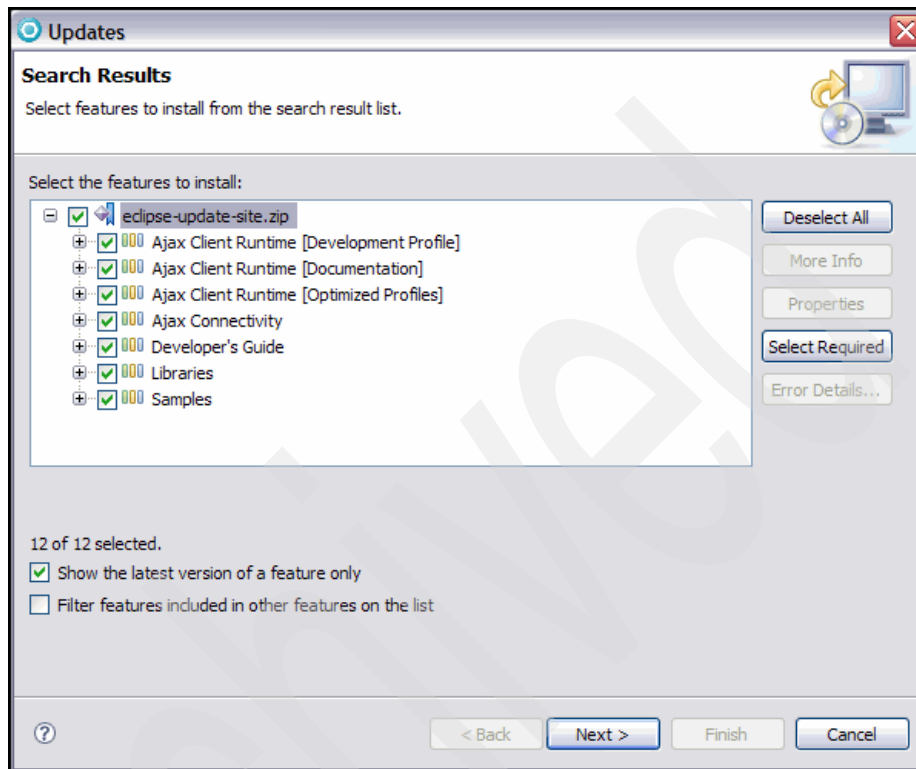


Figure 5-22 List of updates available

- Accept the license agreement and click **Next**.

6. Review the list of features being installed and click **Finish**, as shown in Figure 5-23.

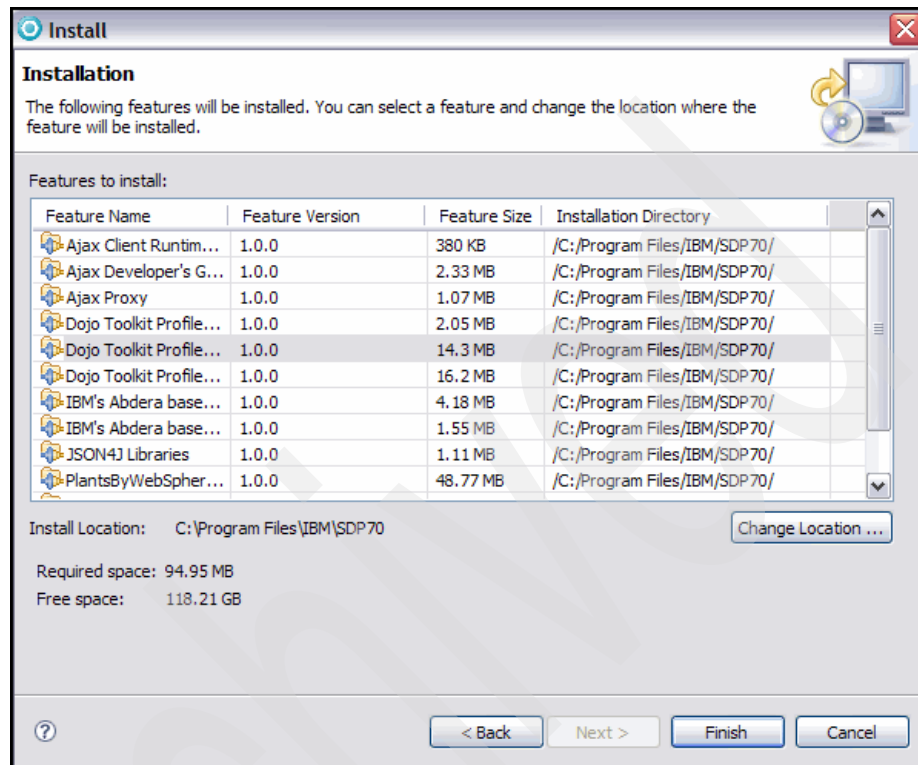


Figure 5-23 Features to be Installed

7. Rational Application Developer will be updated with appropriate Eclipse plug-ins of WebSphere Feature Pack for Web 2.0.
8. Restart Rational Application Developer.

## 5.2.6 Uninstalling IBM Rational Application Developer

IBM Rational Application Developer V7.0 can be uninstalled interactively through IBM Installation Manager. Before uninstallation of any products, ensure that you terminate the programs that you installed using Installation Manager.

1. On the IBM Installation Manager overview page select **Uninstall Packages** (Figure 5-24).
2. On the Uninstall Packages page select the Rational Application Developer product package to uninstall, as shown in Figure 5-24. Click **Next**.



Figure 5-24 Select packages to be uninstalled

3. A summary page is now presented. After reviewing the list of packages that will be uninstalled, click **Uninstall**.
4. Once the uninstall completes, click **Finish** to exit the wizard.

## 5.3 Eclipse 3.3

The Eclipse Project is an open source software development project devoted to creating a development platform and integrated tooling. The Eclipse Platform provides a framework and services that serve as a foundation for tools developers to integrate and extend the functionality of the platform. The platform includes a workbench, concept of projects, user interface libraries (JFace, SWT), built-in help engine, and support for team development and debug. The platform can be leveraged by a variety of software development purposes including

modeling and architecture, integrated development environment (Java, C/C++, Cobol), testing, and so forth.

With a common public license that provides royalty-free source code and world-wide redistribution rights, the Eclipse Platform provides tool developers with great flexibility and control over their software technology. Industry leaders like IBM, Borland, Merant, QNX Software Systems, RedHat, SuSE, TogetherSoft, and WebGain formed the initial eclipse.org board of directors of the Eclipse open source project. More detailed information about Eclipse can be found at:

<http://www.eclipse.org>

Eclipse is packaged in many types of bundles. We have used Eclipse IDE for Java EE Developers, which includes complete support of Java EE 5.

### 5.3.1 Eclipse installation

To install:

1. Download the Eclipse IDE for Java EE Developers from:  
<http://www.eclipse.org/downloads/>
2. Extract the downloaded file eclipse-jee-europa-winter-win32.zip into a local folder on your workstation.

3. Launch Eclipse by double-clicking **eclipse.exe** and then selecting a workspace for the project. Once the Eclipse workbench is open you will see the welcome page, as shown in Figure 5-25.



Figure 5-25 Eclipse welcome page

Once you have successfully installed Eclipse on your workstation you must configure it for the WAS CE server. You can do that in two modes: offline or online.

1. You can download the WAS CE 2.0 WTP Server Adapter update site package from:  
<http://download.boulder.ibm.com/ibmdl/pub/software/websphere/wasce/updates/>
2. Extract this bundle to a local directory.
3. In your Eclipse workbench go to **Help** → **Software Updates** → **Find and Install**.
4. Select **Search for new features to install** and click **Next**.

5. Click **New Local Site**, browse for the folder where we extracted bundle files, and click **OK**, as shown in Figure 5-26.

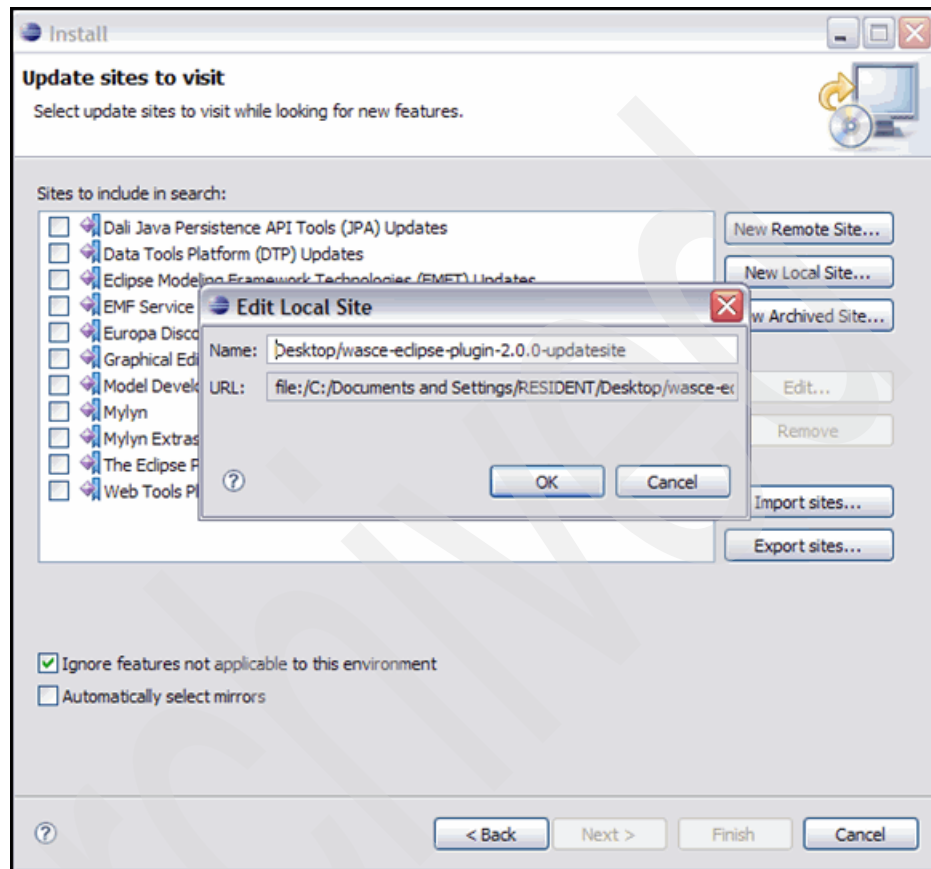


Figure 5-26 Add site location



If you are connected to Internet then you can also choose the option of **New Remote Site**, and can add specify <http://download.boulder.ibm.com/ibmdl/pub/software/websphere/wasce/updates/> as the URL. It will be listed in the Updates Sites to visit section of Eclipse.

1. Select **wasce-eclipse-plugin-2.0.0-updatesite** and click **Finish**, as shown in Figure 5-27.

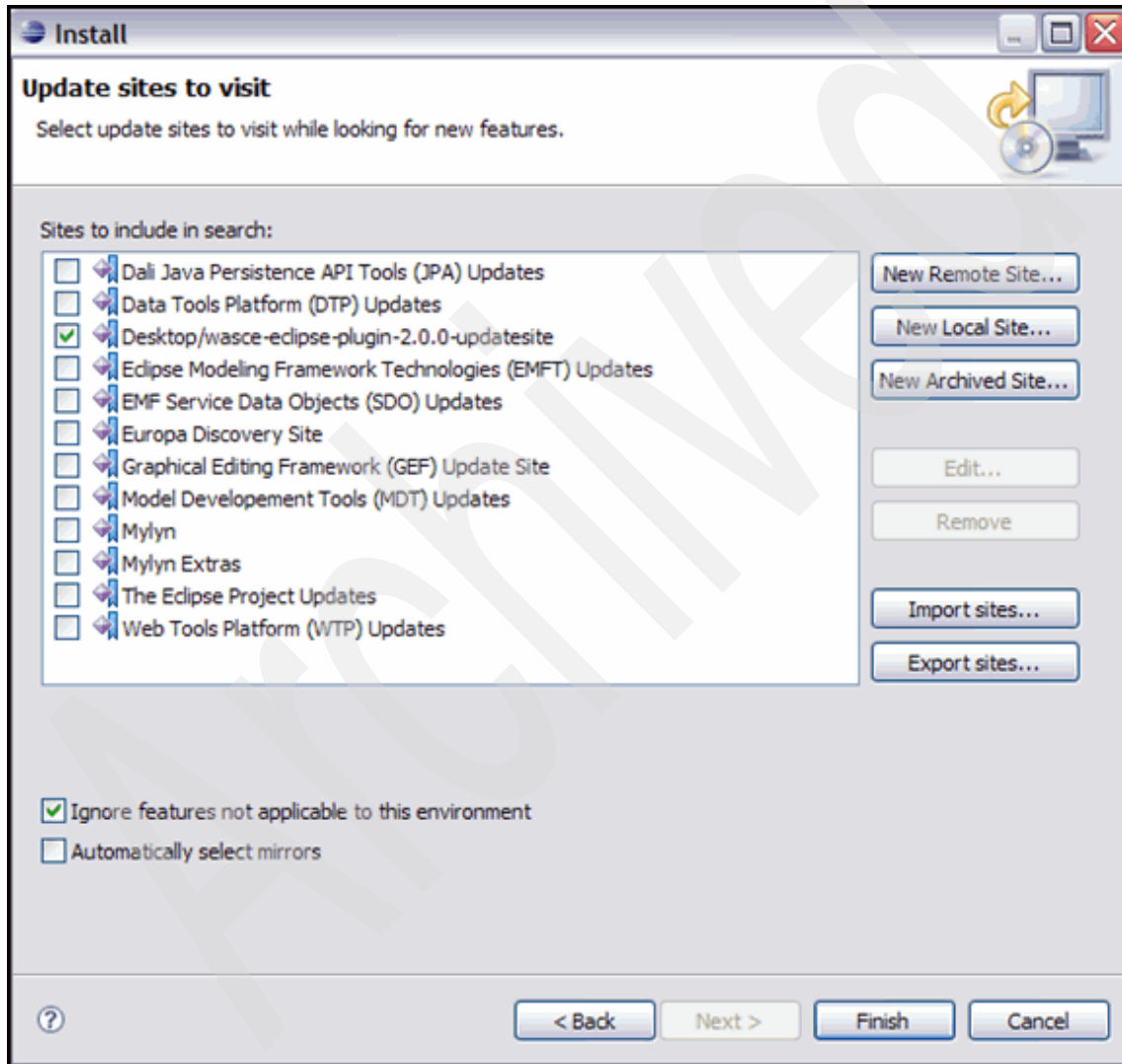


Figure 5-27 Select update sites to look for features

2. A pop-up displays a list of all features available. We must select for WAS CE v2.0 Server Adapter 2.0.0 to install. Click **Next**, as shown in Figure 5-28.

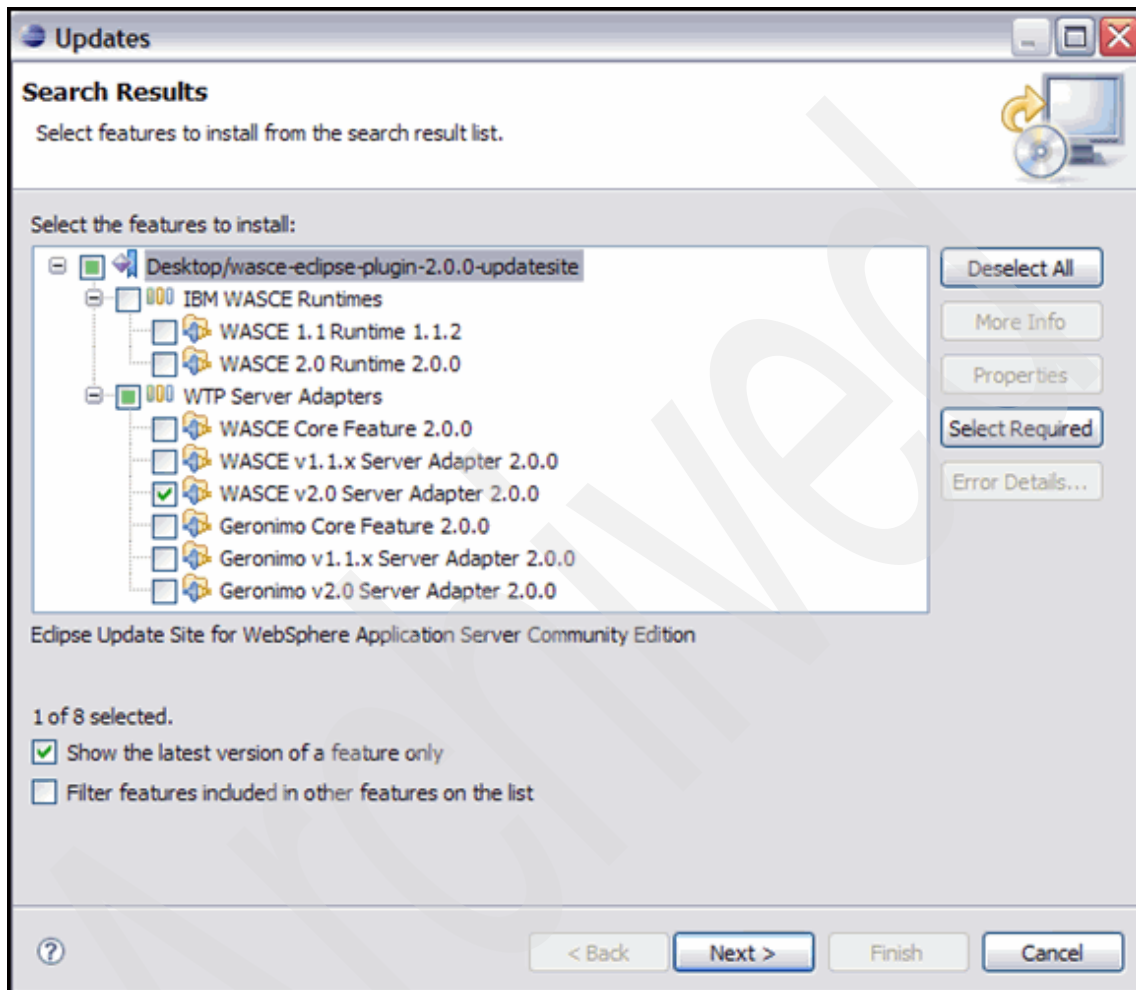


Figure 5-28 Select features to be installed

3. Review the list of features being installed and click **Finish**, as shown in Figure 5-29. This installs the required adapter for WAS CE.

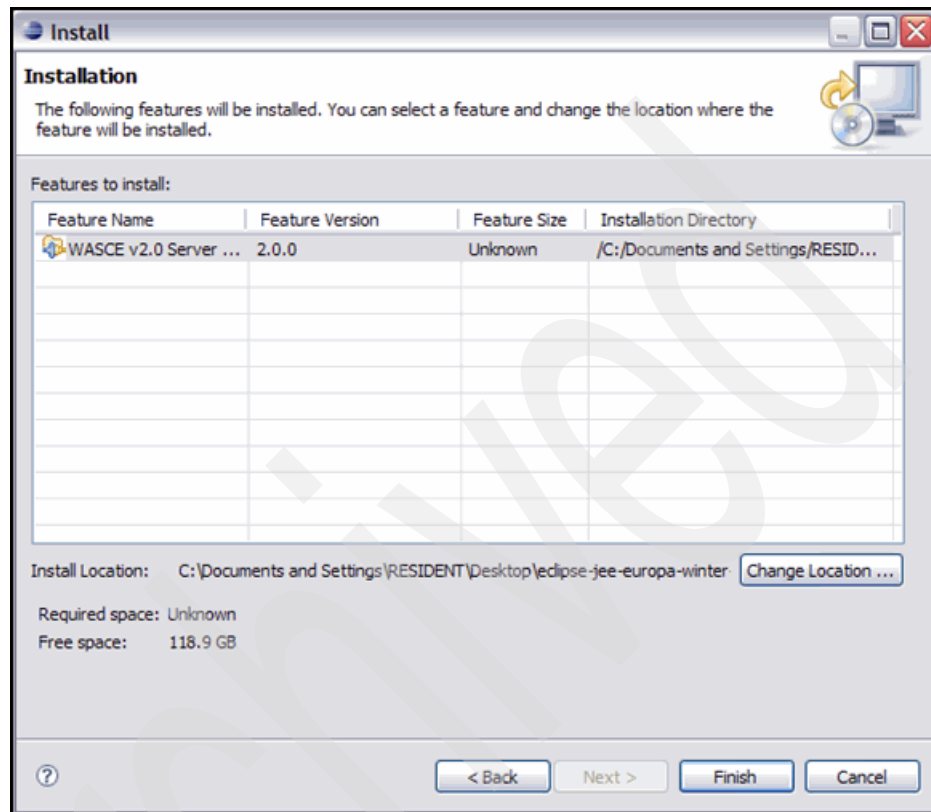


Figure 5-29 Review features being installed

Another way of installing the WAS CE Server adapter is:

1. The WAS CE server adapter can also be installed from the Servers view. Right-click in **Servers View** and select **New** → **Server**.

2. The Define a New Server panel pops up. Click **Download additional server adapters**. This searches for all possible adapters available and displays a list of them, as shown in Figure 5-30.

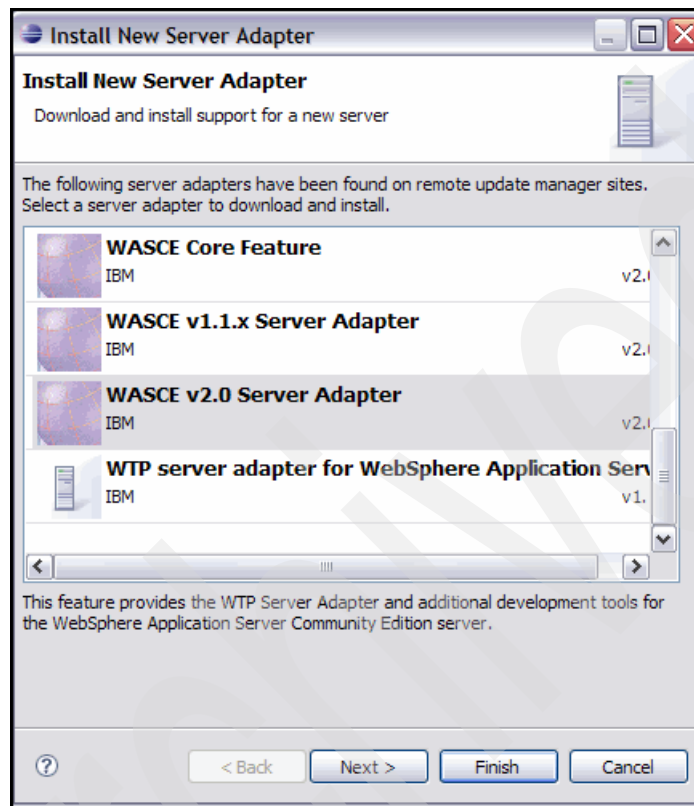


Figure 5-30 Server Adapters available

3. Select **WASCE v2.0 Server Adapter** and click **Next**. A notification server adapter is installed, as shown in Figure 5-31.

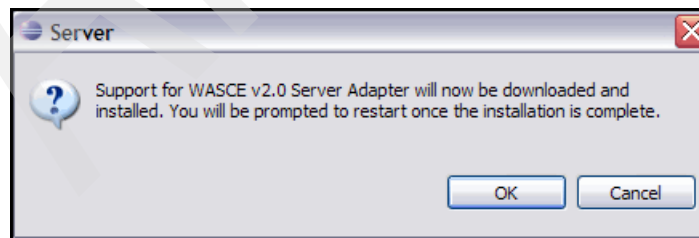


Figure 5-31 Server adapter download start

### 5.3.2 Configuration of Eclipse for WAS CE server

Once the WAS CE Server Adapter is installed you must configure an Eclipse connection to the WAS CE server. To do this:

1. Right-click in the Servers view and **Select New** → **Server**.
2. Expand the IBM folder in the pop-up window and select **IBM WASCE v2.0 Server**. Click **Next**, as shown in Figure 5-32.

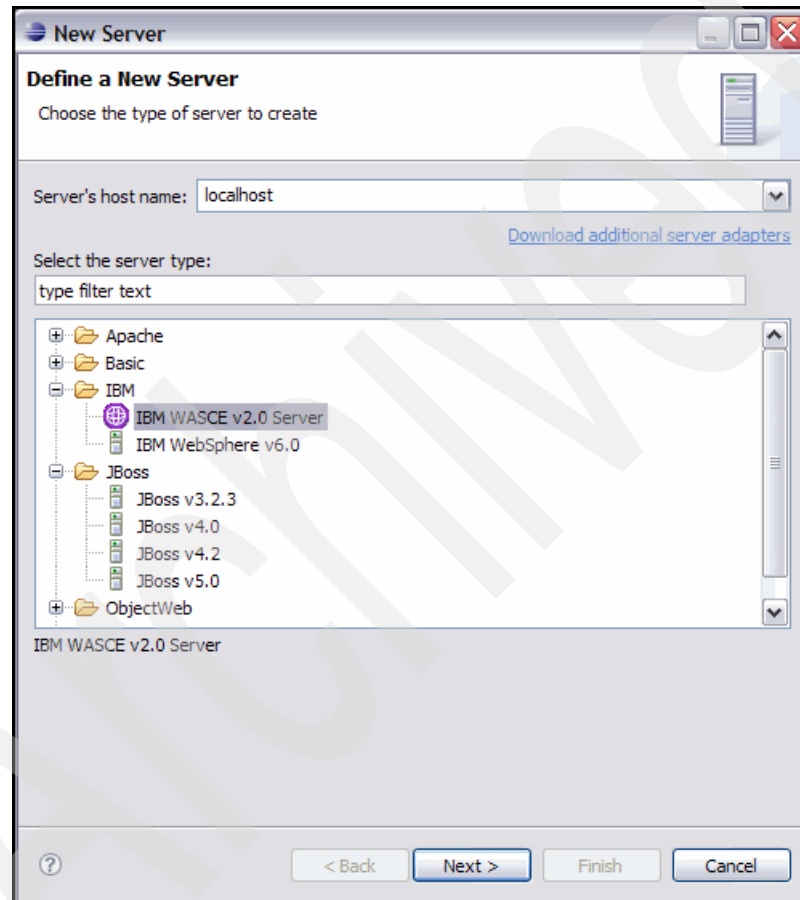


Figure 5-32 Define a new server

3. Browse for the application server installation directory and click **Next**, as shown in Figure 5-33.

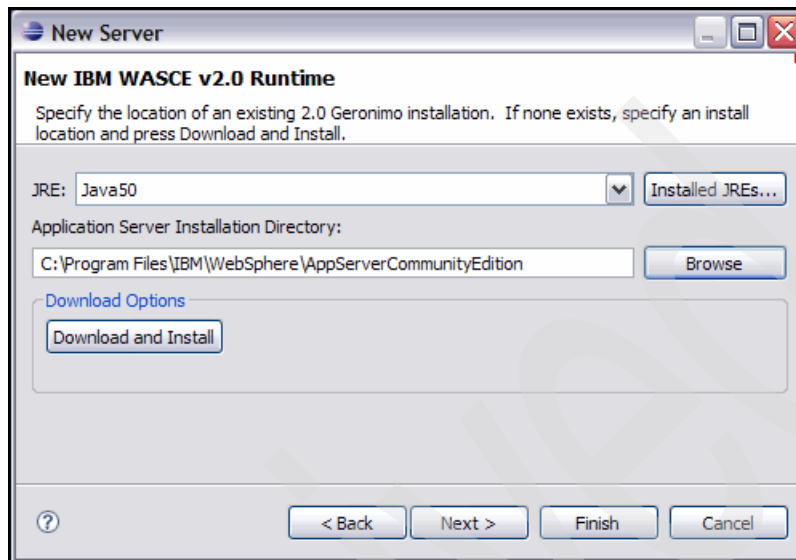


Figure 5-33 WASCE v2.0 runtime location

4. You are attaching a new WASCE server instance in your workbench. The default host name is *localhost*, the default administrator ID is *system*, and the default password is *manager*. The default port assignment for Web Connector is *8080* and the default port assignment for RMI Naming services is *1099*. If any of the defaults are not acceptable simply change the default value before you click **Finish**, as shown in Figure 5-34.

**New Server**

**New IBM WASCE v2.0 Server**

New IBM WASCE v2.0 Server

Hostname:

Administrator id:

Administrator password:

Specify the configured ports for this server.

Port Name	Port Value
Web Connector	8080
RMI Naming	1099

? < Back Next > Finish Cancel

Figure 5-34 WAS CE: Authentication and configuration Information

5. You should now see the IBM WASCE v2.0 Server listed in the Servers view. Right-click this server and select **Start**.
6. The server now starts up and you should see the state change from stopped to started.

The setup of your Eclipse and WAS CE development and testing environment is now complete.

## 5.4 Firebug 1.05

Firebug is one of the most powerful Firefox extensions for JavaScript developers. We can use it to edit, debug, and monitor CSS, HTML, and JavaScript live on any Web page. Firebug is an all-in-one extension that lets you:

- ▶ Debug JavaScript code with breakpoints, with variable monitoring, and by stepping through the code.
- ▶ Review network traffic.
- ▶ Review and tweak the DOM tree and the CSS of the various elements.

The strength and flexibility of Firebug signified a landmark in Ajax development and are good reasons to use Firefox as a preferred development browser. For more information about Firebug go to:

<http://www.getfirebug.com>

### 5.4.1 Installation

Find steps for installation of firebug as add-on to firefox browser:

1. Start the Firefox browser and go to the Mozilla Firefox URL:  
<http://www.mozilla.com>
2. Click **Add-ons** at the top of the page.
3. In the search bar within the Firefox Web site, type Firebug and click **Search**.
4. In the search results, select **Firebug**.
5. Select **Add to FireFox**.
6. In the resulting pop-up dialog box, click **Install now**.
7. Select **Restart Firefox**.
8. In Firefox, click **Tools** → **Firebug** → **Open Firebug**.
9. If you see a message stating that Firebug is disabled at the bottom of the browser window, click **Enable Firebug**.
- 10.) Close the Firefox browser.

From now on, you will have a separate panel that you can activate by clicking the small checkmark in the lower right-hand corner of your browser. Firebug can also be enabled or disabled for selected sites.



## 5.4.2 IE debugging aids

Tools that you may find useful for debugging applications accessed using the Internet Explorer browser are:

- ▶ The IE Developer Toolbar: This lets you better control the IE environment (cache, cookies), view the DOM tree, and explore and manipulate the element's style. This is a useful tool when analyzing the page generated by an application.
- ▶ The Microsoft Script Debugger: This is available with Microsoft Office. It lets you set breakpoints in the JavaScript code, display variables, and perform standard debugging activities. Note that when debugging we recommend that you disable the cache via the IE Developer Toolbar before each debugging session to ensure that you are not debugging a cached version of the code.

IE Developer Toolbar and Microsoft Script Debugger can be downloaded from Microsoft Download Center:

<http://www.microsoft.com/downloads/Search.aspx>

## 5.5 JSON Formatter

JSON Formatter is an online tool helpful in the debugging of Web 2.0 applications when the data is transferred using the JSON format. JSON data is sent without line breaks to minimize the bytes needed to be transferred. However, this feature makes it difficult to read the data when debugging. The JSON Formatter helps developers format the JSON data into a human readable format and can be accessed at the following URL:

<http://curiousconcept.com/jsonformatter/>

In one of our examples data returned from the server in JSON format is as shown in Example 5-1. It is very difficult to understand.

### *Example 5-1 Unformatted JSON*

---

```
{"result":[{"orderList":[{"orderStatus":0,"orderDescription":"Ajax Redbooks","orderNo":"11-123"}, {"orderStatus":0,"orderDescription":"SOA RedBooks","orderNo":"11-124"}], "shippingAddress":{"city":"Gurgaon","state":"HR","street":"Sector - 30"}, {"firstName":"Ankur","email":"ankurgoyal@in.ibm.com","lastName":"Goyal"}, {"orderList":[{"orderStatus":0,"orderDescription":"XML Redbooks","orderNo":"11-125"}, {"orderStatus":0,"orderDescription":"Web 2.0
```

```
RedBooks","orderNo":"11-126"]],"shippingAddress":{"city":"Gwalior","state":"MP","street":"IIITM"},"firstName":"Anuj","email":"goyal.anuj@gmail.com","lastName":"Goyal"}]]}
```

---

But with JSON Formatter it can be presented in a human readable format (Example 5-2).

*Example 5-2 Formatted JSON*

---

```
{
  "result": [
    {
      "orderList": [
        {
          "orderStatus": 0,
          "orderDescription": "Ajax Redbooks",
          "orderNo": "11-123"
        },
        {
          "orderStatus": 0,
          "orderDescription": "SOA RedBooks",
          "orderNo": "11-124"
        }
      ],
      "shippingAddress": {
        "city": "Gurgaon",
        "state": "HR",
        "street": "Sector - 30"
      },
      "firstName": "Ankur",
      "email": "ankurgoyal@in.ibm.com",
      "lastName": "Goyal"
    },
    {
      "orderList": [
        {
          "orderStatus": 0,
          "orderDescription": "XML Redbooks",
          "orderNo": "11-125"
        },
        {
          "orderStatus": 0,
          "orderDescription": "Web 2.0 RedBooks",
          "orderNo": "11-126"
        }
      ]
    }
  ]
}
```

```
    ],  
    "shippingAddress": {  
      "city": "Gwalior",  
      "state": "MP",  
      "street": "IIITM"  
    },  
    "firstName": "Anuj",  
    "email": "goyal.anuj@gmail.com",  
    "lastName": "Goyal"  
  }  
]  
}
```

---



## Ajax connectivity

This chapter discusses Web 2.0 to SOA connectivity for enabling connectivity from Ajax clients to external services or J2EE assets. We introduce the RPC Adapter and the Ajax proxy server along with examples showing implementation details.

This chapter contains the following sections:

- ▶ Web 2.0 to SOA connectivity overview
- ▶ RPC Adapter
- ▶ Ajax Proxy

## 6.1 Web 2.0 to SOA connectivity overview

The Web 2.0 to SOA connectivity provides connectivity from Ajax clients to external Web services, internal SOA services, and other J2EE assets. It also extends enterprise data to customers and partners through Web feeds.

Web 2.0 applications extend the value of SOA through rich user experiences and integration between external and internal content. Due to a number of technical and browser security limitations, basic connectivity between Ajax clients and SOA services has been complex and expensive to develop.

To address these limitations the WebSphere Application Server Feature Pack for Web 2.0 includes an Ajax Proxy component that eliminates the browser's single-origin restriction with cross-domain scripting that combines internal and external services. Additionally, the Web remoting capability simplifies connecting directly to JEE services when extending SOA data outside of the enterprise for partners and customers. See Figure 6-1.

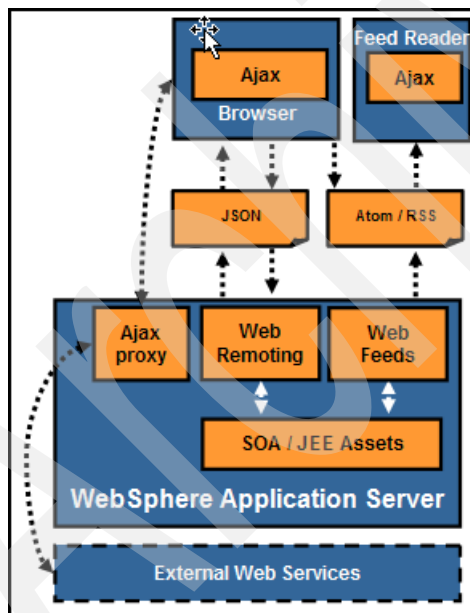


Figure 6-1 Web 2.0 to SOA connectivity

## 6.2 RPC Adapter

Web remoting is a pattern that provides support for JavaScript or client code to directly invoke server logic. This pattern provides the ability to invoke Java methods from JavaScript.

The IBM implementation for Web remoting is referred to as the IBM RPC Adapter. The RPC Adapter is designed to help developers create command-based services quickly and easily in a manner that complements the programming styles applicable to Ajax applications and other lightweight clients. Implemented as a generic servlet, the RPC Adapter provides an HTTP interface to registered Java Beans.

Web remoting provides a lightweight Web endpoint that can expose methods of Java EE assets (EJB, POJO, Web service proxies). Web remoting is easily invoked from Ajax applications using JSON or XML formats and supports HTTP GET/POST mapping for methods. It is enabled through simple configuration options without requiring modifications to the original Java objects, EJB, or Web services. See Figure 6-2.

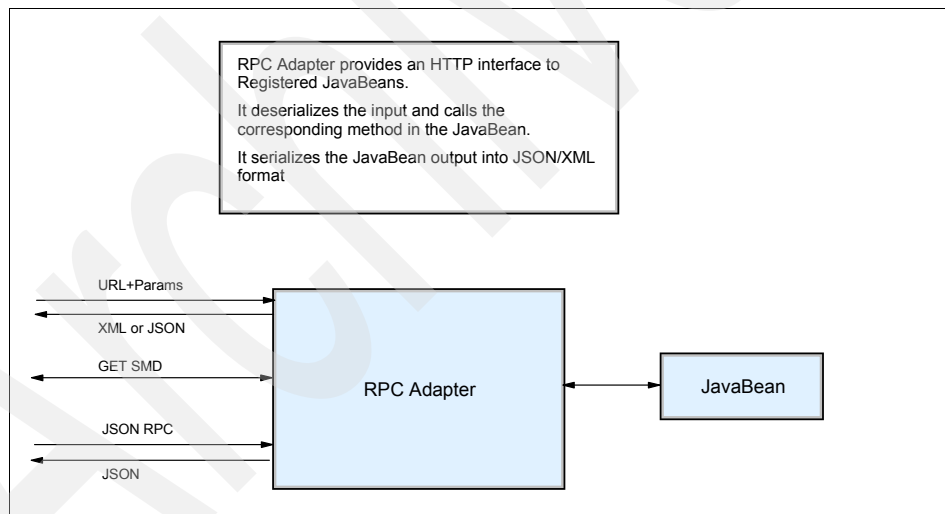


Figure 6-2 RPC Adapter architecture

## 6.2.1 Protocols supported by RPC Adapter

The RPC Adapter currently supports two RPC protocols:

- ▶ HTTP - RPC

In HTTP RPC, invocations are made using URLs with query parameters or form parameters. The RPC Adapter intercepts and then deserializes the URL to obtain the service name, method name, and input parameters. Using this information, the RPC Adapter then invokes the corresponding method of the registered Java Beans.

- ▶ JSON - RPC

In JSON-RPC, method invocation is accomplished using JSON objects. The response generated is also a JSON object. The registered Java Beans can be accessed through the Dojo JSON-RPC API.

## 6.2.2 Sample RPC Adapter implementation

In this chapter we use the CourierApp sample application as an example of implementing the RPC Adapter with Java POJO classes in a step-by-step fashion.

1. Start the Rational Application Developer in a new workspace. Close the Welcome page and switch to the Web perspective.



2. Click **File** → **Project** in the workbench toolbar. Select **Dynamic Web Project** in the Web folder and click **Next**, as shown in Figure 6-3.

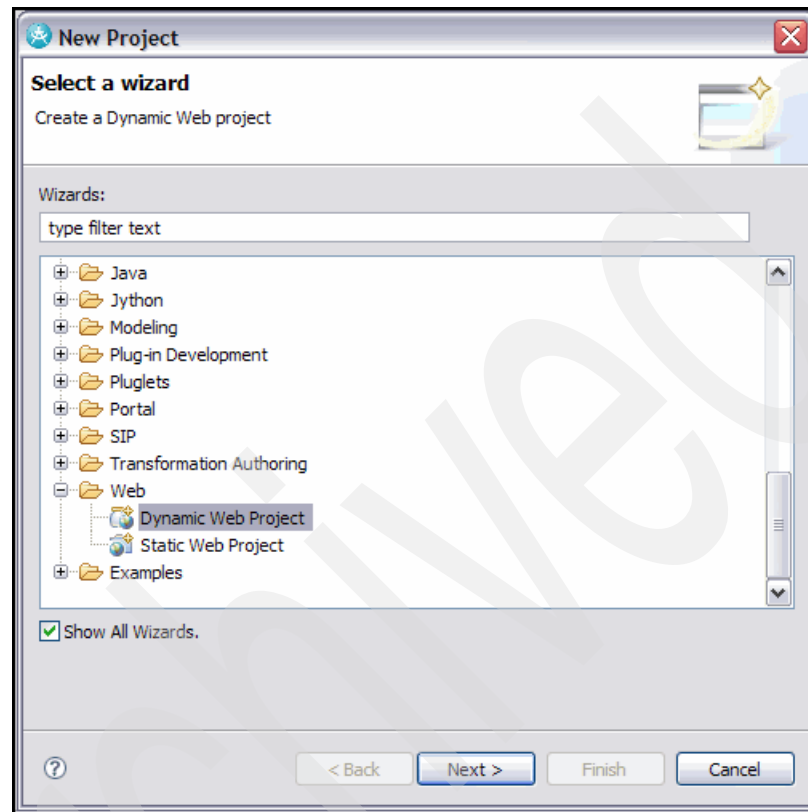


Figure 6-3 New Project wizard

3. Enter the project name as ItsorPCAdapter, the target run time as **WebSphere Application Server 6.1**, check the **Add project to an EAR** check box, and click **Finish**, as shown in Figure 6-4.

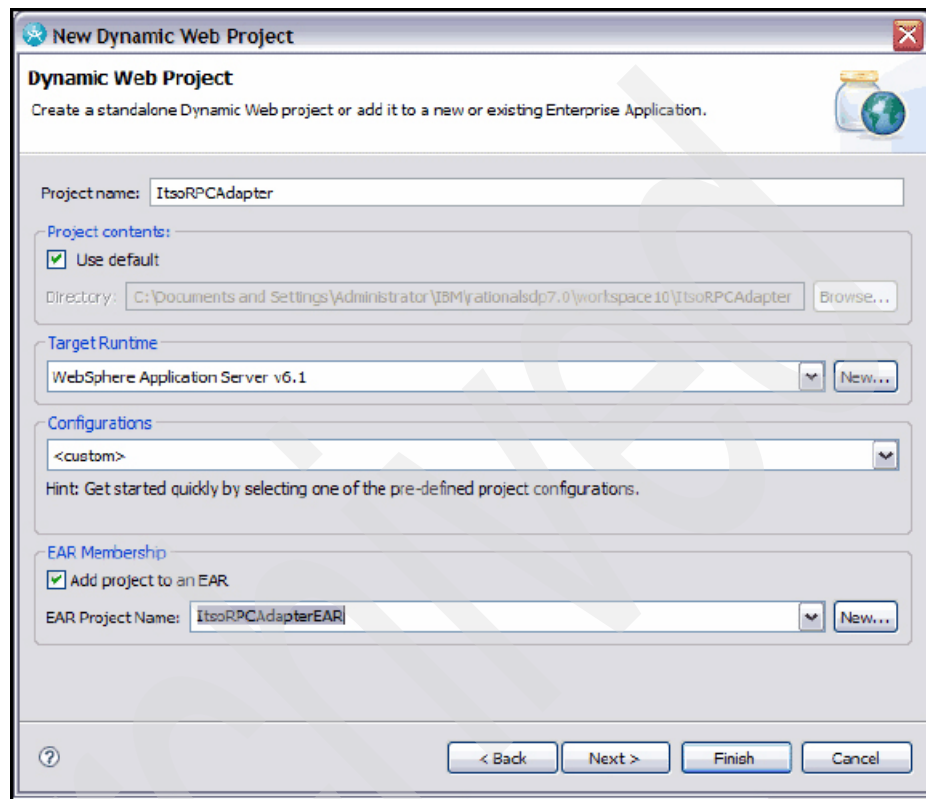


Figure 6-4 Dynamic Web Project wizard

4. In the Project Explorer browse through the Web project structure, as shown in Figure 6-5. Right-click the WEB-INF/lib folder and select **import**.

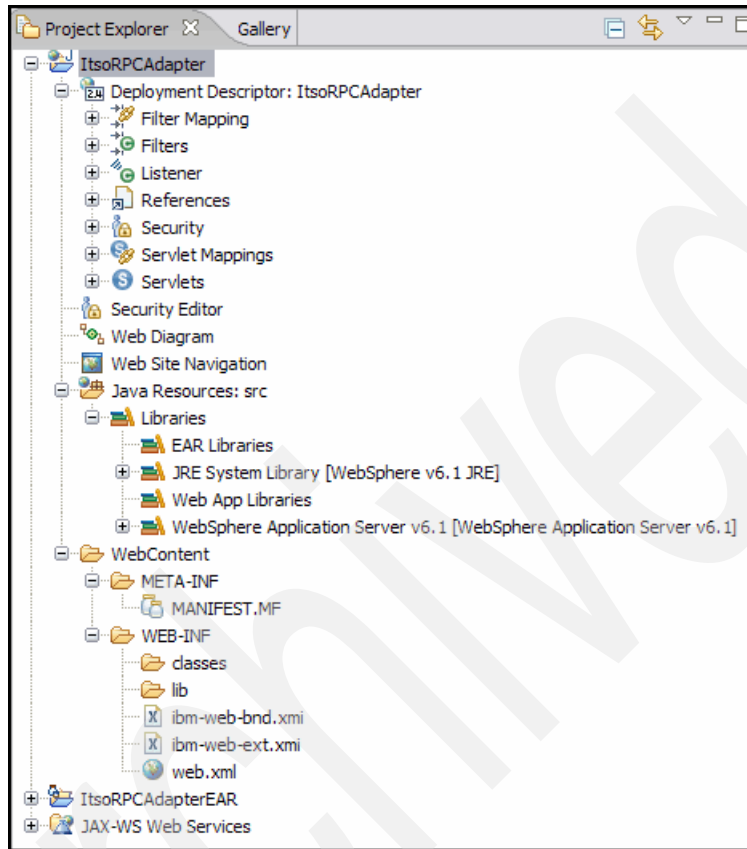


Figure 6-5 Web project structure

5. Select **General** → **File System** in the Import dialog and click **Next**. Browse to the <<Installation Directory>>\SDP70\runtimes\base\_v61\Web2fep\optionalLibraries\ folder. Within the folder select the **RPCAdapter.jar** file, the **retroweaver-rt-2.0.jar** file, and the **commons-logging-1.0.4.jar** file and click **Finish**, as shown in Figure 6-6.

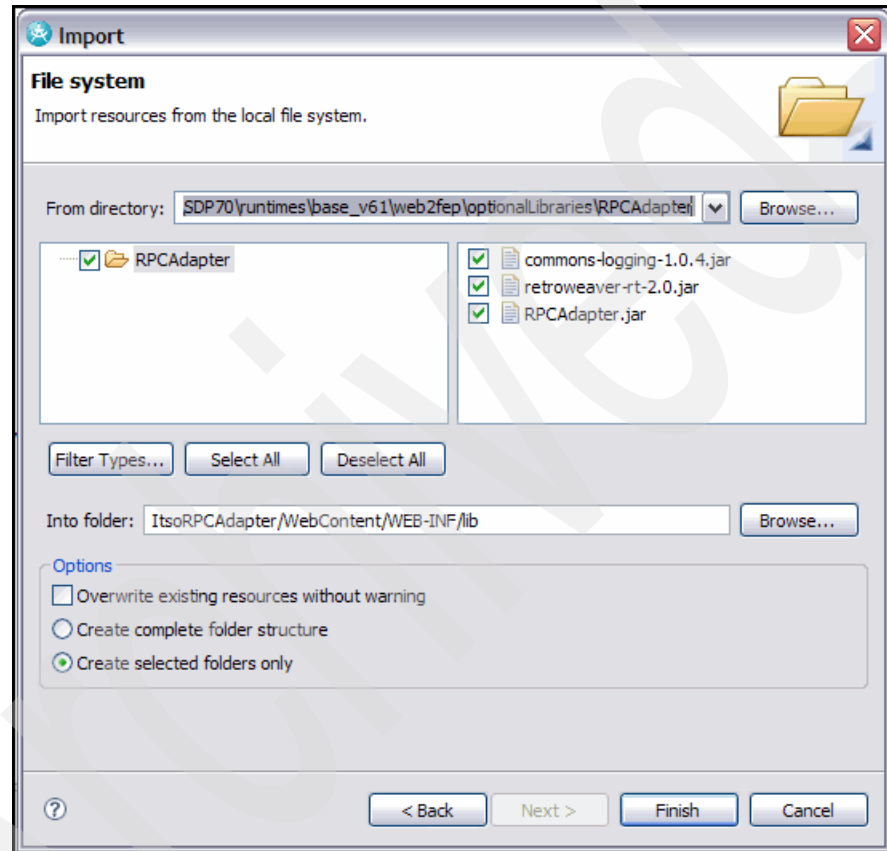


Figure 6-6 Import supporting jar files

6. Similarly, import the JSON.jar file from <<Installation Directory>>\SDP70\runtimes\base\_v61\Web2fep\optionalLibraries\JSON4J. All of these library files can be verified from the Web App Libraries in the Libraries folder. The JSON.jar, along with other jar files, is required by the RPCAdapter library at run time.
7. In the Project Explorer view open the **WEB-INF\Web.xml** file by double-clicking it. It should open in the Deployment Descriptor editor window.

8. Select the **Servlets** tab at the bottom of the editor and click **Add**, as shown in Figure 6-7.

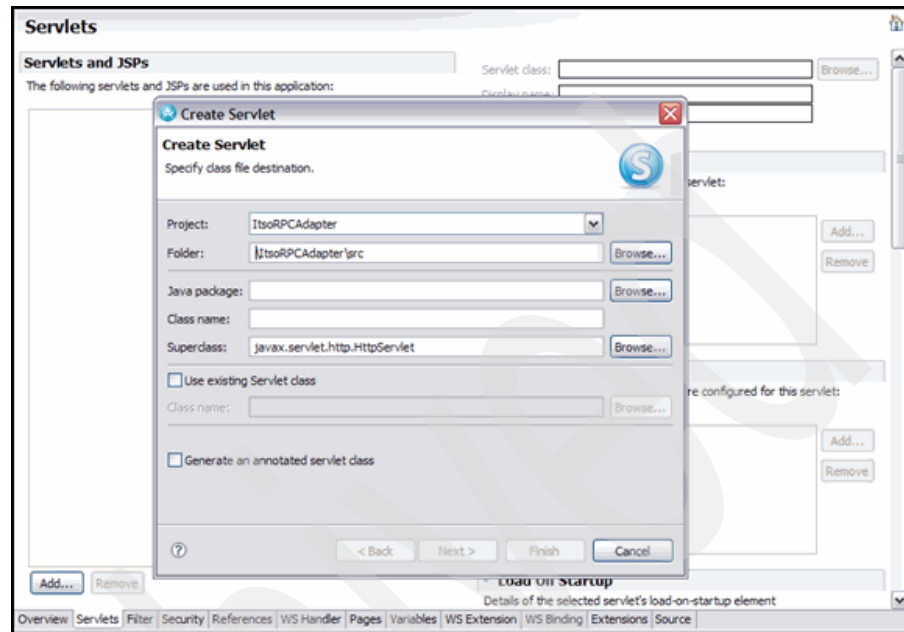


Figure 6-7 Create Servlet page

9. Select **Use existing Servlet Class**, browse for the **com.ibm.websphere.rpcadapter.RPCAdapter**, class and click **Next**.

10. Add **/RPCAdapter/\*** in the URL Mapping section and click **Finish**, as shown in Figure 6-8.

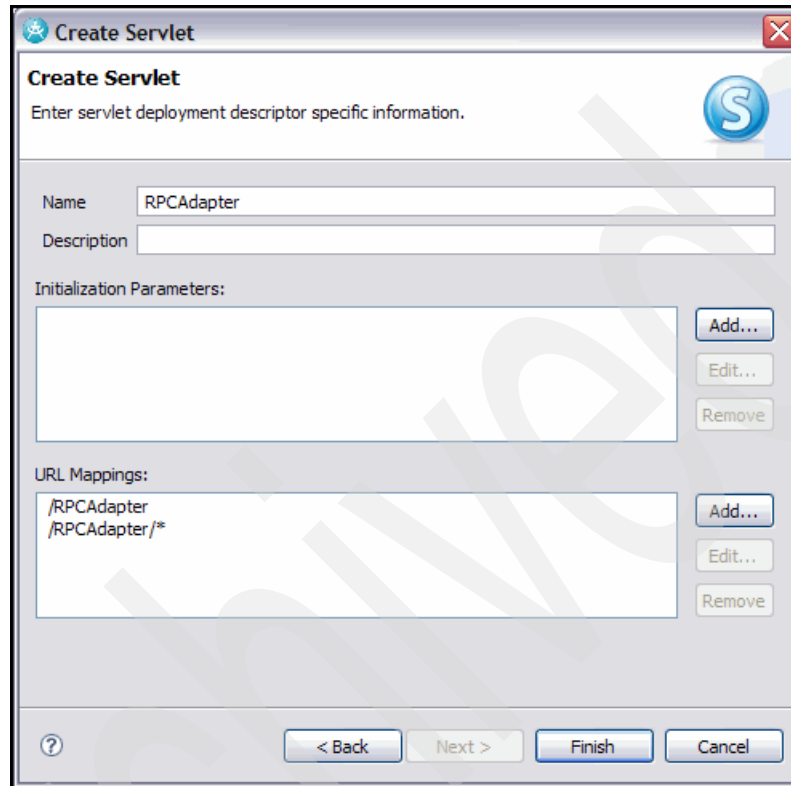


Figure 6-8 Create Servlet: *RPCAdapter* URL Mapping

This step configures the Web.xml file so that the servlet `com.ibm.websphere.rpcAdapter` is exposed under the URL:

`http://<host>:<port>/<contextRoot>/RPCAdapter/*`

All requests with *RPCAdapter* in the URL are routed to the `com.ibm.websphere.rpcadapter.RPCAdapter` servlet.

11. From the Deployment Descriptor editor click the **Source** tab of Web Deployment Descriptor and add the configuration parameters shown in Example 6-1 to the Web.xml file.

*Example 6-1 Web deployment descriptor*

```
<servlet>
  <description>
  </description>
```

```
<display-name>RPCAdapter</display-name>
<servlet-name>RPCAdapter</servlet-name>

<servlet-class>com.ibm.websphere.rpcadapter.RPCAdapter</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>RPCAdapter</servlet-name>
  <url-pattern>/RPCAdapter</url-pattern>
</servlet-mapping>
<servlet-mapping>
  <servlet-name>RPCAdapter</servlet-name>
  <url-pattern>/RPCAdapter/*</url-pattern>
</servlet-mapping>
```

---

12. Right-click the **WEB-INF** folder in the Project Explorer and select **New** → **File**. In the Update File Name text box enter `RpcAdapterConfig.xml` and click **Finish**.
13. All POJOs that will be exposed through the RPCAdapter must be specified in this configuration file.

14. Start the application server and from the Servers view add the ItsorPCAdapter project by right-clicking the server in the Servers view and selecting **Add and Remove Project**. Click **Finish**, as shown in Figure 6-9.

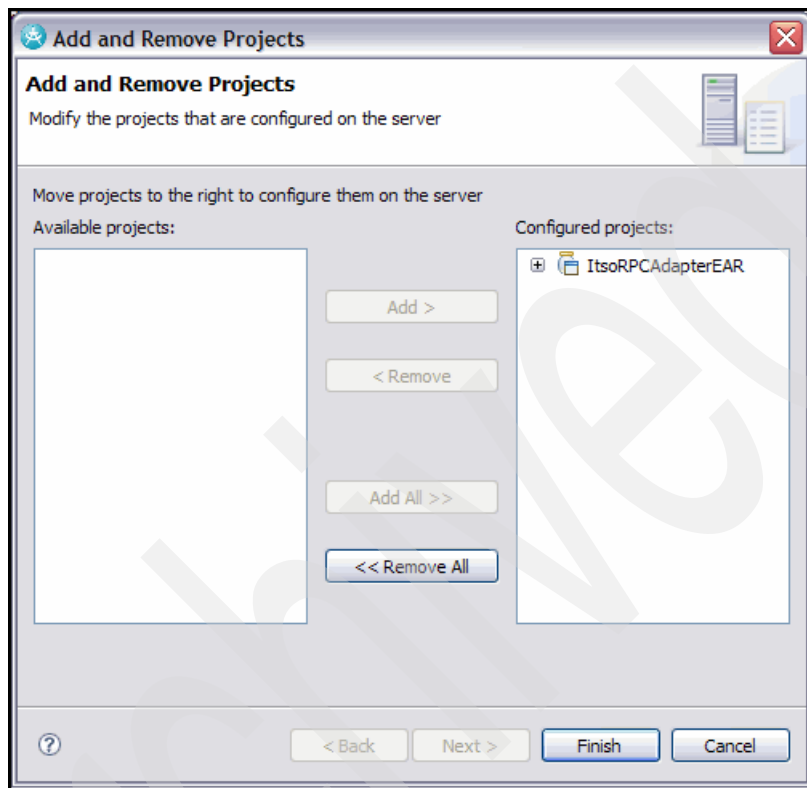


Figure 6-9 Add and remove project to publish on test environment



15. For our example purposes we have added four Java classes in the `com.ibm.itso.billing` package that will be exposed as POJO (Figure 6-10):
- **Customer:** This represents customer information and includes name, e-mail, phone, shipping and billing addresses, and a list of associated orders.
  - **Address:** This represents address information in terms of street, city, and state and is used to represent the billing and shipping address information in customer class.
  - **Order:** This represents the order information in terms of OrderNo, description, order date, and status and is associated with customer information.
  - **CustomerList:** This represents a POJO object with two instances of customer information and exposes the public methods being mapped to the URL with the help of the RPC Adapter.

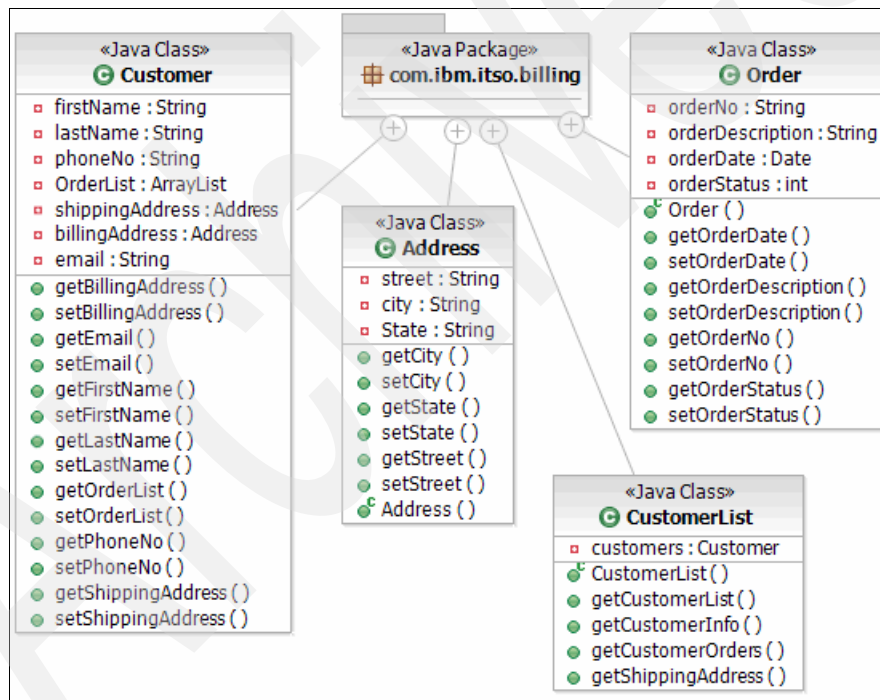


Figure 6-10 Package structure for this section

16. Once all the classes are added we must configure the WEB-INF/RPCAdapterConfig.xml to add the POJO configuration parameters, as shown in Example 6-2.

*Example 6-2 RpcAdapterConfig.xml*

---

```
<?xml version="1.0" encoding="UTF-8"?>
<rpcAdapter
xmlns="http://www.ibm.com/xmlns/prod/websphere/featurepack/v6.1/RpcAdapterConfig"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <default-format>xml</default-format>
  <services>
    <pojo>
      <name>customerlist</name>

<implementation>com.ibm.itso.billing.CustomerList</implementation>
      <methods filter="whitelisting">
        <method>
          <name>getCustomerList</name>
          <description>Displays customer List.</description>
        </method>
      </methods>
    </pojo>
  </services>
</rpcAdapter>
```

---

RpcAdapterConfig.xml is an XML configuration file. All parameters are within <rpcAdapter>...</rpcAdapter> tags.

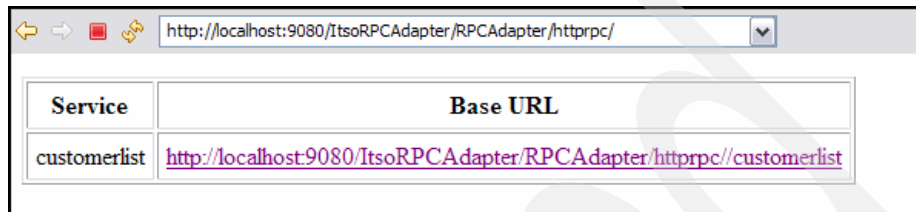
- <default-format>: This can be XML or JSON depending on the type of output required.
  - <pojo>: This is used for mapping to the CustomerList object with name, implementation, and methods elements.
  - <name>: This is the name of the mapped instance of object.
  - <implementation>: This is the implementation details of the POJO being exposed.
  - <methods>: This is a list of all methods being exposed with the filter attribute.
  - <method>: This is the information about a particular method with the name, description, and parameters.
17. Once the configuration of the RpcAdapterConfig.xml is complete right-click the **ItsoRPCAdapter** project in Project Explorer and select **RunAs** → **Run on server**.

18.in the Run On Server pop-up select **Next** and then **Finish**.

19.From a browser window set the address bar to:

`http://localhost:9080/ItsORPCAdapter/RPCAdapter/httprpc/`

A list of published services and their URLs will be returned, as shown in Figure 6-11

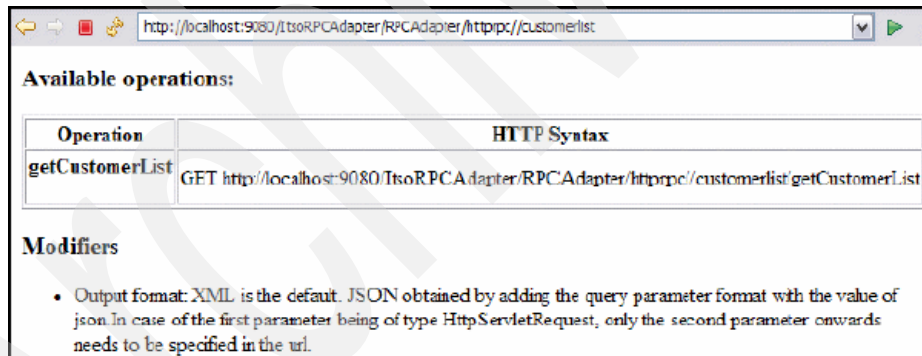


Service	Base URL
customerlist	<a href="http://localhost:9080/ItsORPCAdapter/RPCAdapter/httprpc//customerlist">http://localhost:9080/ItsORPCAdapter/RPCAdapter/httprpc//customerlist</a>

Figure 6-11 Output of the httprpc request: customerlist POJO mapped to URL

20.In the browser output click the link

**`http://localhost:9080/ItsORPCAdapter/RPCAdapter/httprpc//customerlist`**, which displays list of all the methods exposed, as shown in Figure 6-12.



Operation	HTTP Syntax
getCustomerList	GET <a href="http://localhost:9080/ItsORPCAdapter/RPCAdapter/httprpc//customerlist/getCustomerList">http://localhost:9080/ItsORPCAdapter/RPCAdapter/httprpc//customerlist/getCustomerList</a>

**Modifiers**

- Output format: XML is the default. JSON obtained by adding the query parameter format with the value of json. In case of the first parameter being of type HttpServletRequest, only the second parameter onwards needs to be specified in the url.

Figure 6-12 POJO operations mapped to URL

21.Set the address bar in browser to the URL:

`http://localhost:9080/ItsORPCAdapter/RPCAdapter/httprpc//customerlist/getCustomerList`

View the result from the CustomerList object in XML format, as shown in Figure 6-13.



Figure 6-13 Result of POJO operation in XML

You can expand shippingAddress, billingAddress, and orderList to see detailed information.

22. In the RpcAdapterConfig.xml file change the <default-format> value to json. and repeat the previous steps to publish the project to the server.

23. In the browser set the address bar to the URL:

http://localhost:9080/ItsoRPCAdapter/RPCAdapter/httprpc/customerlist/getCustomerList

The result should now be displayed in JSON format, as shown below.

```
{
  "result": [
    {
      "orderList": [
        {
          "orderStatus": 0,
          "orderDate": {
            "date": 8,
            "day": 2,
            "timezoneOffset": -330,
            "year": 108,
            "month": 3,
            "hours": 21,
            "seconds": 9,
            "minutes": 24,
            "time": 1207670049750
          },
          "orderDescription": "Ajax Redbooks",
          "orderNo": "11-123"
        },
        {
          "orderStatus": 0,
          "orderDate": {
            "date": 8,
            "day": 2,
            "timezoneOffset": -330,
            "year": 108,
            "month": 3,
            "hours": 21,
            "seconds": 9,
            "minutes": 24,
            "time": 1207670049750
          },
          "orderDescription": "SOA RedBooks",
          "orderNo": "11-124"
        }
      ],
      "shippingAddress": {
        "city": "Gurgaon",
        "state": "HR",
        "street": "Sector -
```

```

30"},"firstName":"Ankur","billingAddress":{"city":"New
Delhi","state":"Delhi","street":"Sector - 13
Dwarka"},"email":"ankurgoyal@in.ibm.com","lastName":"Goyal","phoneNo
":"981851931"},{"orderList":[{"orderStatus":0,"orderDate":{"date":8,
"day":2,"timezoneOffset":-330,"year":108,"month":3,"hours":21,"secon
ds":9,"minutes":24,"time":1207670049750},"orderDescription":"XML
Redbooks","orderNo":"11-125"},{"orderStatus":0,"orderDate":{"date":8
,"day":2,"timezoneOffset":-330,"year":108,"month":3,"hours":21,"seco
nds":9,"minutes":24,"time":1207670049750},"orderDescription":"Web
2.0
RedBooks","orderNo":"11-126"}],"shippingAddress":{"city":"Gwalior","
state":"MP","street":"IIITM"},"firstName":"Anuj","billingAddress":{"
city":"SGNR","state":"RAJ","street":"Jawahar
Nagar"},"email":"goyal.anuj@gmail.com","lastName":"Goyal","phoneNo":
"1542464164"}]}

```

Other methods are mapped into the `RpcAdapterConfig.xml` file by adding their elements following `<method>` elements, as shown in Example 6-3.

*Example 6-3 Mapping additional methods into the `RpcAdapterConfig.xml` file*

---

```

<method>
  <name>getCustomerInfo</name>
  <description>Displays customerInformation.</description>
  <parameters>
    <parameter>
      <name>firstName</name>
      <description>Contains first Name of Customer.</description>
    </parameter>
  </parameters>
</method>
<method>
  <name>getCustomerOrders</name>
  <description>Displays customer Orders information.</description>
</method>
<method>
  <name>getShippingAddress</name>
  <description>Displays customer Shipping address
information.</description>
</method>

```

---

Notice that all parameters for a method are specified using the `<parameter>` tag for a method being exposed.

24. Publish the project to the application server and display the list of methods available as described previously in step 20 on page 111.

25. Set the address bar in your browser to:

`http://localhost:9080/ItsoRPCAdapter/RPCAdapter/httprpc/customerlist/getCustomerInfo?firstName=ankur`

The results have been filtered as per the method parameter specified in the `RpcAdapterconfig.xml` file, as shown in Figure 6-14.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <results>
-   <Customer>
+     <shippingAddress>
+     <orderList>
      <firstName>Ankur</firstName>
+     <billingAddress>
      <email>ankurgoyal@in.ibm.com</email>
      <lastName>Goyal</lastName>
      <phoneNo>981851931</phoneNo>
    </Customer>
  </results>
```

Figure 6-14 Result of POJO operation

Try calling other methods with appropriate parameters and observe the results.

### 6.2.3 RPC Adapter library features

In this section we discuss RPC Adapter library features.

#### White listing and black listing

We can white list or black list a set of methods in a POJO service. White and black listing are done using the `filter` attribute of the `methods` element in the POJO element. The values for the filter can be *whitelisting* or *blacklisting*. If the filter attribute is not specified then all the methods are listed.

We have been using whitelisting as the filter value. Therefore, all the methods mapped in the `RpcAdapterConfig.xml` file are published and accessible.

#### Example 1

Remove the filter attribute of the `methods` element in the `RpcAdapterConfig.xml` file and then check the list of published methods as described in step 20 on page 111. You will observe some new methods like `equals`, `hashCode`, `toString`, and so on, in the list, as all of these methods are inherently part of `CustomerList` POJO.

### Example 2

Modify the attribute *filter* value to *blacklisting* and again check the list of published methods.

Observe that none of the methods configured in the `RpcAdapterConfig.xml` file are published and accessible.

### Example 3

Remove the method `getShippingAddress` from the `RpcAdapterConfig.xml` file and check the list of published methods. Observe that the `getShippingAddress` method is no longer blacklisted and is now accessible.

## Getting access to `HttpServletRequest` in a POJO method

In many cases you need to get access to an `HttpServletRequest` object in the methods being called from JavaScript. This can be accomplished using the RPC Adapter by specifying the `HttpServletRequest` as the first parameter in the method call. The SMD returned does not contain the `HttpServletRequest` as a parameter. However, you can call the method from JavaScript without passing it as a parameter. For example, consider the method `putNameInSession(HttpServletRequest req, String name)`. The corresponding XML follows, as shown in Example 6-4.

Example 6-4 *putNameInSession* method mapping

---

```
<services>
<POJO>
<methods filter="whitelisting">
  <method>
    <name>putNameInSession</name>
    <description>Puts a name in session</description>
    <http-method>POST</http-method>
    <parameters>
      <parameter>
        <name>req</name> <!-- Should be exactly the parameter name in the
java method-->
        <description>The HttpServletRequest object</description>
      </parameter>
      <parameter>
        <name>name</name> <!-- Should be exactly the parameter name in the
java method-->
        <description>The name entered by the user</description>
      </parameter>
    </parameters>
  </method>
</methods>
```

```
</POJO>  
</services>
```

---

## Validators

Validators are defined by specifying the validators elements in the RPC Adapter configuration file. You can specify a set of validators for individual POJO services. Before a method is invoked, the validate methods are called for parameters that refer to at least one validator. All of the validators should extend the abstract class `com.ibm.websphere.rpcadapter.Validator`.

Another way of validating is by using regular expressions. The `validation-regex` element can be used to specify regular expressions that should match the parameter values. If the parameter values do not match the regular expression then a validation error occurs. Validation errors result in an error object being returned in either JSON or XML format. The most significant fields of this error object are `name`, `code`, `message`, `methodName`, `serviceName`, `parameterIndex`, and `parameterValue`.

We add two validators to validate the input parameters of the `getCustomerOrders` and the `getShippingAddress` methods. One validator is a default validator for string inputs and the other is for the email input validation. Add the elements shown in Example 6-5 to the `RpcAdapterConfig.xml` file to configure validators after the `<default-format>` element.

### *Example 6-5 Adding validators*

---

```
<validators>  
  <validator id="default">  
    <validation-regex>([A-Za-z])+</validation-regex>  
    <validation-class>com.ibm.websphere.rpcadapter.DefaultValidator</validation-class>  
  </validator>  
  
  <validator id="emailid">  
    <validation-regex>^[a-zA-Z][\w\.-]*[a-zA-Z0-9]@[a-zA-Z0-9][\w\.-]*[a-zA-Z0-9]\.[a-zA-Z][a-zA-Z\.-]*[a-zA-Z]$</validation-regex>  
    <validation-class>com.ibm.websphere.rpcadapter.DefaultValidator</validation-class>  
  </validator>  
</validators>
```

---



Two validators are defined with a validator-ref of default and emailid. To bind these validators with parameters we must use the <validator-ref> element following the <description> element of each parameter. See Example 6-6.

*Example 6-6 Binding validators to parameters*

---

```
<method>
  <name>getCustomerOrders</name>
  <description>Displays customer Orders information.</description>
  <parameters>
    <parameter>
      <name>firstName</name>
      <description>Contains First Name of Customer.</description>
      <validator-ref>default</validator-ref>
    </parameter>
  </parameters>
</method>
<method>
  <name>getShippingAddress</name>
  <description>Displays customer Shipping address
information.</description>
  <parameters>
    <parameter>
      <name>email</name>
      <description>Contains email id of Customer.</description>
      <validator-ref>emailid</validator-ref>
    </parameter>
  </parameters>
</method>
```

---

After updating the RpcAdapterConfig.xml file publish it to the application server and set the address bar in your browser to:

<http://localhost:9080/ItsRPCAdapter/RPCAdapter/httprpc/customerlist/getCustomerInfo?firstname?@ankur>

The parameter validator associated with the firstName parameter will validate the input and display an error message in either XML or JSON format, as shown in Figure 6-15.



Figure 6-15 Error returned because of incorrect parameter

Error message CWRPC0009E was received since the parameter value of @ankur is considered invalid because the regular expression applied as the default validator allows only characters and will not allow special characters such as the @ in this case.

Now repeat by setting the browser address bar to the following:

http://localhost:9080/ItsorPCAdapter/RPCAdapter/httprpc/customerlist/getCustomerInfo?firstName=ankur

You should see the successful results of the method.

Try calling the `getShippingAddress` method with an invalid e-mail address and observe the error message, as shown in Figure 6-16.

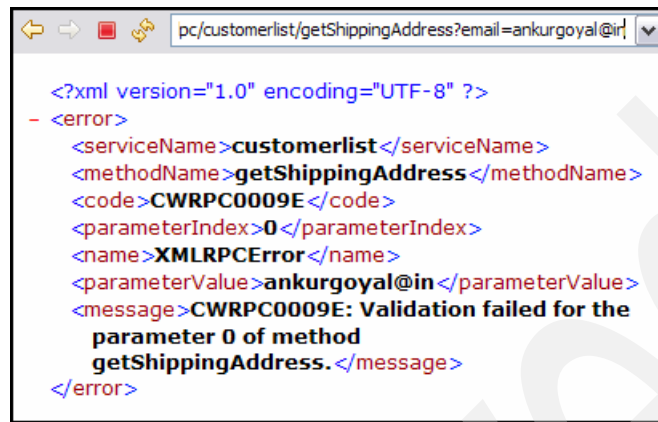


Figure 6-16 Error returned because of incorrect email parameter

In this case the emailid validator is checking the input parameter for a correct value.

Multiple validators can be specified by using a comma-separated list. You can also specify a `validator-ref` tag at the JavaBean level for applying the validator to all the arguments of all the methods. The JavaBean validation can be overridden by defining validators at the parameter level.

## Object scopes

The RPC Adapter supports scopes for the objects that it exposes. There are three different scopes supported by RPC Adapter, namely, request, session, and application. Specifying these scopes results in the exposed object being stored as request, session, and application, respectively. An additional advantage when the scope is set to application or session is that the object is not recreated each time but is looked up using the session or application scope. The key used is of the format "com.ibm.websphere.rpcadapter:*ServiceName*", where *ServiceName* is the name of exposed object. In our case we specified a session scope of customerlist for the <pojo> service, as shown in Example 6-7.

*Example 6-7 Object scopes*

---

```
<services>
  <pojo>
    <name>customerlist</name>

    <implementation>com.ibm.itso.billing.CustomerList</implementation>
    <scope>Session</scope>
    <methods filter="whitelisting">
```

---

The object created from com.ibm.itso.billing.CustomerLit will be put in a session with a key com.ibm.websphere.rpcadapter:customerlist", where customerlist is the serviceName.

## Return values in session

The RPC Adapter supports enabling return values to be added to the session. To enable this the `<add-to-session>` tag needs to be included. In our case, if we want to add the result of `getCustomerInfo` into the session we can configure it as shown in Example 6-8.

*Example 6-8 Return values in session*

---

```
<method>
  <name>getCustomerInfo</name>
  <description>Displays customerInformation.</description>
  <parameters>
    <parameter>
      <name>firstName</name>
      <description>Contains first Name of Customer.</description>
      <validator-ref>default</validator-ref>
    </parameter>
  </parameters>
  <add-to-session>customerInformation</add-to-session>
</method>
```

---

Once this method is specified in a URL the result of `customerinformation` will be added in the session and is accessible during the session. The value of the `add-to-session` tag is the key under which the return value of the method will be put in the session.

## Complex object support

The RPC Adapter supports complex objects, for example, objects that can contain other objects. Complex objects are supported as return types as well as method parameters. The RPC Adapter can serialize any type of complex object. Note that maps and collections are not supported as parameters to method calls. This is because the class of the objects in maps and collections cannot be determined from its contents. Support for this requires class hinting or parameterized types, which is not yet supported. Also, you cannot call methods with complex parameters via HTTP-RPC even though complex return types are supported. In our case we are returning `customerlist`, which is a complex object. It contains information about shipping and billing addresses that is an instance of `address` and also the array `OrderList` that includes instances of `order` class.

## Recursive object support

This represents a particular case of complex object support, for example,  $A \rightarrow B \rightarrow A$  or  $A$  contains  $B$ , which contains  $A$ . The IBM RPC Adapter component handles this by replacing the duplicate occurrences of objects with `$jref` (JSON

serialization) or \$xref (XML Serialization) placeholders. These place holders contain information that can be used to look up the original object (that is, an XPath expression in the case of XML and a JavaScript expression in the case of JSON). The support for recursive object handling can be enabled by setting the value of the <recursive-object-support> tag to true, as shown in Example 6-9.

*Example 6-9 recursive-object-support*

---

```
<rpcAdapter>
<default-format>xml</default-format>
<filtered>false</filtered>
<recursive-object-support>true</recursive-object-support>
<validators>
<validator id="default">
  <validation-regex>([A-Za-z])+</validation-regex>

  <validation-class>com.ibm.websphere.rpcadapter.DefaultValidator</validationclass>
</validator>
---
---
---
</rpcAdapter>
```

---

### **Serialization of selected attributes**

Often the objects exposed contain certain fields that the application developer does not want to expose via the JSON or XML Web service. The RPC Adapter provides the ability to suppress fields from the returned object. This is enabled by means of the tag <serialized-params>. To eliminate the phoneNo attribute of Customer add the <suppressed-fields> tag, as shown in Example 6-10.

*Example 6-10 serialized-params*

---

```
<serialized-params>
  <serialized-param>
    <serialized-param-type>
      com.ibm.itso.billing.Customer
    </serialized-param-type>
    <suppressed-fields>
      <suppressed-field>phoneNo</suppressed-field>
    </suppressed-fields>
  </serialized-param>
</serialized-params>
```

---

The phoneNo attribute will be eliminated from the return object customerlist. Another use of <suppressed-fields> is to suppress objects of one class from being serialized irrespective of which class contains it. To configure the RPC Adapter to remove order object completely in the returned, add these tags to the RpcAdapterConfig.xml file, as shown in Example 6-11.

*Example 6-11 Removing order object completely*

---

```
<serialized-params>
  <serialized-param>
    <serialized-param-type>
      com.ibm.itso.billing.Order
    </serialized-param-type>
    <suppress>true</suppress>
  </serialized-param>
</serialized-params>
```

---

### **Aliases for classes**

You can specify aliases for classes. The alias is used as a node name during the XML serialization. We added two aliases for order and customer class in the RpcAdapterConfig.xml file, as shown in Example 6-12.

*Example 6-12 Aliases for classes*

---

```
<serialized-params>
  <serialized-param>
    <serialized-param-type>
      com.ibm.itso.billing.Order
    </serialized-param-type>
    <alias>customerorder</alias>
  </serialized-param>
  <serialized-param>
    <serialized-param-type>
      com.ibm.itso.billing.Customer
    </serialized-param-type>
    <alias>customer</alias>
  </serialized-param>
</serialized-params>
```

---

After publishing the updated Web application to the application server you should see the results shown in Figure 6-17.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <results>
  - <customer>
    + <shippingAddress>
    - <orderList>
      + <customerorder>
      + <customerorder>
    </orderList>
```

Figure 6-17 Aliases declared for customerorder and customer

Instead of the <entry> and <orderEntry> tags the results are in the <customer> and <customerorder> tags as configured in the RpcAdapterConfig.xml file.

## Convertors for classes

A user can specify convertors for JSON or XML formats. If a convertor is specified for a class, the convertor is used for the JSON or XML serialization of that class. A convertor class must implement the `com.ibm.websphere.rpcadapter.converters.IConverter` interface. By default the RPC Adapter comes with the following convertors:

- ▶ `com.ibm.websphere.rpcadapter.converters.sql.Date` (used to convert `java.sql.Date` to date and time string)
- ▶ `com.ibm.websphere.rpcadapter.converters.sql.Time` (used to convert `java.sql.Time` to date and time string)
- ▶ `com.ibm.websphere.rpcadapter.converters.sql.TimeStamp` (used to convert `java.sql.TimeStamp` to date and time)
- ▶ `com.ibm.websphere.rpcadapter.converters.util.Date` (used to convert `java.util.Date` to date and time string)

In the case of the `orderDate` attribute in the `order` object, to covert it into an `RpcAdapter` format, add the <converter> tags, as shown in Example 6-13.

Example 6-13 Class convertors

```
<default-format>xml</default-format>
<converters>
  <converter>
    <bean-class>java.util.Date</bean-class>
    <converter-class>
      com.ibm.websphere.rpcadapter.converters.util.Date
    </converter-class>
```



```
</converter>
</converters>
```

---

After applying these tags the orderDate is converted into the format shown in Example 6-14.

*Example 6-14 Converted orderDate attribute*

```
<orderDate>2008-04-09T23:16:21+0530</orderDate>
```

---

i

```
- <orderDate>
  <date>9</date>
    <day>3</day>
    <year>108</year>
    <timezoneOffset>-330</timezoneOffset>
    <month>3</month>
    <hours>23</hours>
    <seconds>27</seconds>
    <minutes>26</minutes>
    <time>1207763787750</time>
  </orderDate>
```

*Figure 6-18 Result before converter is defined*

## Enabling JSON content filtered output

For security reasons the RPC Adapter can be configured to omit the JSON comments from the filtered output when used for exposing methods as JSON Web services. For example, JSON data wrapped with '/' and '\*'. Dojo has the ability to process the comment filtered JSON at the client side. By default this feature is turned off in the RPCAdapter. It is enabled by setting the filtered tag to true in the RpcAdapterConfig.xml file, as shown in Example 6-15.

*Example 6-15 Enabling JSON content filtered output*

```
<rpcAdapter>
  <default-format>xml</default-format>
  <filtered>true</filtered>
  -----
  -----
</rpcAdapter>
```

---

## Security

Security support in the RPC Adapter is achieved by using J2EE Web security. All the services have their own unique URLs. Access to those URLs is restricted by the J2EE Web security. This involves creating a security realm in the application server and then defining role-based access to the URLs in the deployment

descriptor file and the Web.xml file, then mapping these roles to users or groups in the security-realm using server-specific configuration.

## 6.2.4 Accessing EJBs through RPC Adapter

For accessing an EJB, first create a wrapper class around the EJB and then expose this wrapper class as a POJO service through the RPC Adapter.

## 6.2.5 XML schema for RPCAdapterConfig.xml

Complete XML schema definition for RPCAdapterConfig.xml file is available in Appendix A, “XML schema definitions” on page 645.

## 6.2.6 Limitations of the RPC Adapter libraries

The limitations are:

- ▶ POJOs registered with the RPC Adapter must not contain argument constructors that are declared public. The RPC Adapter instantiates the POJOs, and then invokes the specified method in response to RPC requests.
- ▶ When a parameter is specified for a particular method in the .xml file, you must specify all of the parameters for that method in the.xml file.
- ▶ There is no built-in authentication and authorization support. However, you can use J2EE security for authorization and authentication, as there is a URL corresponding to each of the services that are exposed.
- ▶ There is no support for passing maps and collections as parameters to the called methods. They are supported as return types provided that their keys are strings or integers. Object keys in maps are not supported.
- ▶ The HTTP-RPC protocol only supports primitives and simple objects with constructors of type ctor (string).

## 6.2.7 The CourierApp sample

This section describes the CourierApp sample in order to demonstrate the features of the RPC Adapter. Five topics are covered:

- ▶ Overview: Overview of the CourierApp sample
- ▶ Prerequisites: Prerequisites for the CourierApp sample
- ▶ Limitations: Limitations of the CourierApp sample
- ▶ Installation: Installing the CourierApp sample
- ▶ Usage: Usage of the CourierApp sample

## 6.2.8 Overview of the CourierApp sample

The CourierApp sample Web application is a sample that demonstrates the capabilities of RPC Adapter-like handling of XML and JSON responses, white list and black list methods, and specifying validators. The steps for the installation of the CourierApp sample are on the Installation page. You can access these instructions by using the link <http://yourhost:yourport/CourierApp>. This brings up a page that has links to courier service using JSON or XML. If you select the **Courier Service using JSON** link, the corresponding page is displayed. After selecting a zip or postal code, the address details are automatically populated.

This uses RPC Adapter's JSON service to access the address lookup service. The same application is also implemented using RPC Adapters XML RPC. For example, access:

[http://yourhost:yourport/CourierApp/CourierService\\_xmlrpc.jsp](http://yourhost:yourport/CourierApp/CourierService_xmlrpc.jsp)

The sample courier application is also available as an Eclipse plug-in. After importing the plug-in you can import the courier application to your Eclipse workspace by selecting **File** → **New** → **Example** → **IBM WebSphere Application Server Feature Pack for Web 2.0 Samples** → Courier Application RPC Adapter sample.

## 6.2.9 CourierApp sample prerequisites

The prerequisites for installing the CourierApp sample are shown in Table 6-1.

Table 6-1 Prerequisites

Courier application prerequisites	Version
Java2 Technology Edition	1.4 or 5.0
J2EE1.4 or JEE 5 application server	WebSphere Application Server version 6.x or IBM WebSphere Application Server Community Edition version 2.x

## 6.2.10 Limitations of the CourierApp sample

The CourierApp sample application does not expose all of the features of the RPC Adapter.

## 6.2.11 Installing the CourierApp sample application

This section describes the steps to be followed to locate and install the CourierApp sample application on WebSphere Application Server Version 6.x and WebSphere Application Server Community Edition Version 2.x, as well as the steps to import it as an example in Eclipse. Let us assume that <install\_root> is the directory where the feature pack is installed in the case of WebSphere Application Server (for example, the directory where WebSphere Application Server is installed) and <WASCEFEP\_HOME> is the directory where IBM WebSphere Application Server Community Edition Version 2.0 Plug-in for Web 2.0 Version 1.0 is unzipped.

- ▶ Locating the stand-alone sample: The CourierApp sample for the RPCAdapter library included in the IBM WebSphere Application Server Feature Pack for Web 2.0 is located in <install\_root>/Web2fep/samples/RPCAdapter/CourierApp.war when installed for WebSphere Application Server v6.x. In the case of WebSphere Application Server Community Edition v2.x the libraries can be located at <WASCEFEP\_HOME>/CourierApp/samples/CourierApp.war.
- ▶ Locating the Eclipse example plug-in: The Eclipse archived update site for the IBM WebSphere Application Server Feature Pack for Web 2.0 is provided at the <install\_root>/Web2fep/site directory and will be a zip file. Importing this into Eclipse automatically imports the Eclipse example plug-in for the CourierApp sample. The Eclipse plug-in is not provided in the feature pack for WebSphere Application Server Community Edition v2.x.

The steps to deploy the sample are given below.

### WebSphere Application Server Community Edition 2.x

To deploy this:

1. Start WebSphere Application Server Community Edition 2.x.
2. After the server has started, open the browser and navigate to <http://yourhost:yourport/console>, e.g. <http://localhost:8080/console>.
3. Enter the user name as system and password as manager and log into the administrative console.
4. In the left navigation pane, click the **Deploy New** link.
5. Browse to the directory where you extracted the sample and select the sample in the Archive text box.
6. Click **Install** to deploy the CourierApp sample.
7. Browse to <http://yourhost:yourport/CourierApp> to bring up the CourierApp Welcome page (for example, <http://localhost:8080/CourierApp>).

## WebSphere Application Server V6.x

For deploy:

1. Start the WebSphere Application Server V6.x.
2. Navigate to the Administration console. The default URL is `http://yourhost:yourport/ibm/console` (for example, `http://localhost:9060/ibm/console`).
3. Enter the user name and password that you configured and log in to the administration console.
4. From the left navigation pane select the **Install New Application** link under the Applications section.
5. Select **Browse** and select the CourierApp.war file. Enter the context root as CourierApp.
6. In the context root textbox, enter a name for the application, for example, CourierApp.
7. Click **Next** → **Next** → **Continue**. (Not all of these steps may be applicable for WAS 6.1.)
8. Click **Next** until you reach the summary page and then click **Finish**.
9. After the installation is complete, click the **Save** link to directly save to the master configuration.
10. From the left navigation pane select the **Enterprise Applications** link under the Applications section.
11. Select **CourierApp\_war** and select **Start** to start the sample application.
12. Navigate to `http://yourhost:yourport/yourContextRoot` to bring up the CourierApp Welcome page, for example, `http://localhost:9080/CourierApp`.

## Eclipse example

The steps for installing the CourierApp in an Eclipse environment are as follows:

1. Start Eclipse. The plug-in requires Eclipse 3.2.x with WTP 1.5.x.
2. Select **Help** → **Software Updates** → **Find and Install**.
3. Select **Search for new features to install**.
4. On the Install panel, select **New Archived Site**.
5. Enter the location of the IBM WebSphere Application Server Feature Pack for Web 2.0 Eclipse archive zip file. This zip file should be located in the `<install_root>/Web2fep/site` directory.
6. Complete the plug-in installation. You are prompted to restart Eclipse when you finish.

7. When Eclipse restarts go to **File → New → Example → IBM WebSphere Application Server Feature Pack for Web 2.0 Samples** and select the **Courier Application RPC Adapter sample** and click **Next**.
8. Select the **Target** run time and click **Next → Finish**. This imports the CourierApp sample application as a Web project in Eclipse.
9. You can then deploy the application to the target run time as you would deploy any other Web project. Note that the context root should be given as CourierApp.

Browse to `http://yourhost:yourport/CourierApp` to bring up the welcome page, for example, `http://localhost:8080/CourierApp`.

The host name and ports may vary according to your server version and its configuration.

## Using the CourierApp sample application

CourierApp demonstrates auto population of address details on selecting a zip/postal code. Address lookup service is invoked using the RPC Adapter to get the detailed address. The response can be in either JSON or XML format.

Figure 6-19 shows the starting page of the CourierApp sample.

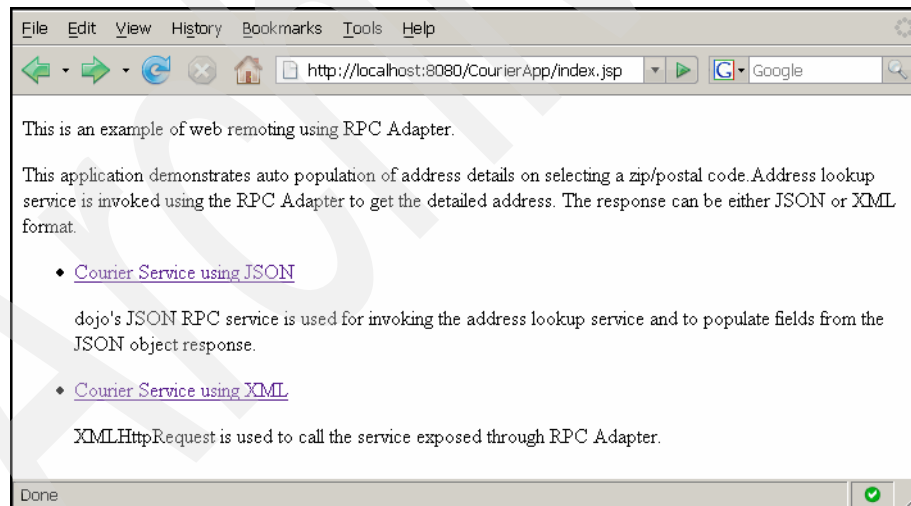


Figure 6-19 CourierApp sample starting page

Figure 6-20 shows the application itself. Once you choose a zipcode from the drop-down box a request to the server is sent and the line 2, 3, and 4 text boxes are automatically populated with the data returned in either a JSON or XML format.

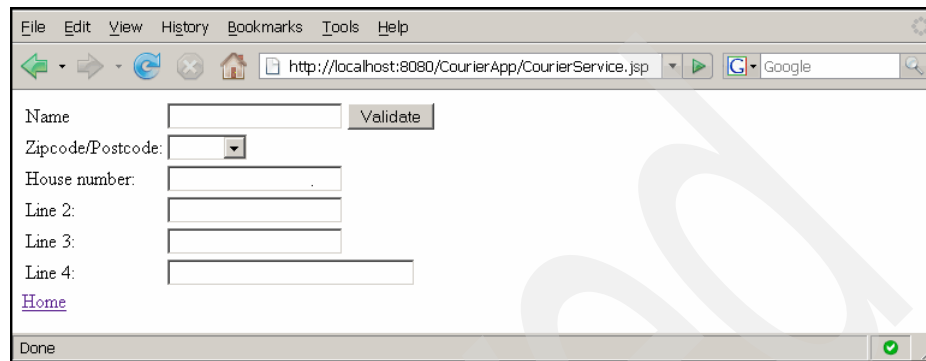


Figure 6-20 CourierApp sample application UI

## 6.3 Ajax Proxy

In this section we discuss the Ajax Proxy.

### 6.3.1 Why we need it

Ajax-based Web applications may need to make Ajax requests to servers other than the server that served the HTML document. For example, your company intranet portal Ajax application requires a feed from the [www.cnn.com](http://www.cnn.com) Web site.

Ajax communication methods include the XMLHttpRequest and the IFrame requests. These methods enable the browser to send HTTP requests to a server at any time with or without a user action. An IFrame or XMLHttpRequest request is restricted from making a request to a server other than the one that served the original HTML page. This limitation is sometimes known as a same domain limitation or same-origin policy and exists as a security measure to prevent hacker script injection by redirecting the page to an untrusted server. However, an Ajax-based Web application might need to make a request to a server that is different from the server that served the main HTML page. Client-side methods exist to work around the same domain limitation, but these methods have drawbacks. The recommended solution to the same-domain limitation is to use a proxy server to forward the request to a server on a different domain.

An Ajax proxy enables browser-based access to cross-site services. It can run embedded within a Web Archive (WAR) file embedded within another J2EE application or as a stand-alone application. In addition, it uses J2EE application-level security for proxy access control and does not require the WebSphere run time or configuration integration since its configuration is static. Ajax proxy also supports white-listing policies for filtering criteria of incoming requests such as cookies, mime types, HTTP headers, and HTTP verbs such as GET, POST, and PUT.

### **6.3.2 How it is implemented**

In a network environment, a proxy is positioned between the requesting client and the server. It accepts requests from the client and passes them onto the server. It then receives the response from the server and passes the response back to the requesting client.

An Ajax proxy server is an application-level proxy server that mediates HTTP requests and responses between Web browsers and servers. Ajax proxies allow Web browsers to bypass the same-origin policy restriction and access cross-domain vendor servers using the XMLHttpRequest.



As an example, if the JavaScript application originated from domain A and attempts to use an XMLHttpRequest to domain B, the browser will prevent the domain B request. The proxy can be used to broker the request, which provides the appearance to the client that the request came from the same server from which the JavaScript originated. See Figure 6-21.

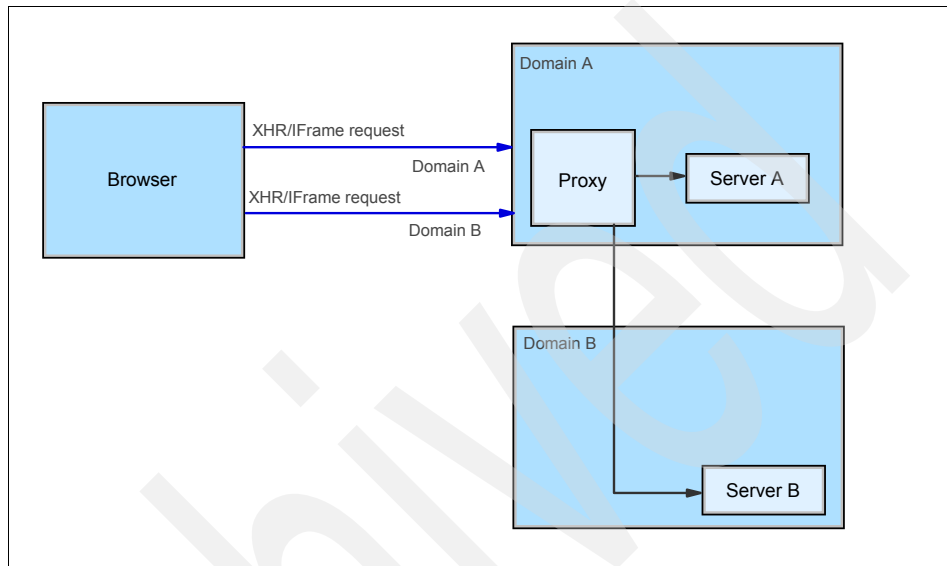


Figure 6-21 Ajax Proxy architecture

The proxy provided with IBM WebSphere Application Server Feature Pack for Web 2.0 is a reverse proxy installed near one or more servers. Connections coming through the reverse proxy are forwarded to the requested server. From the client's perspective, it appears that the requests are originating from the same server even though the reverse proxy might forward requests to different Web servers. The proxy can be used to broker client requests from multiple domains while using Ajax. JavaScript sand-boxing rules prevent network requests to servers other than where the JavaScript originated from.

### 6.3.3 Ajax Proxy architecture and configuration

Figure 6-22 shows the general architecture of an Ajax Web application that implements the Ajax proxy. Notice that different endpoints aggregate to the same domain making it appear to the client that all communication is with a single endpoint. The Ajax proxy is customized with the application and it runs within the Web container. Therefore, performance is determined by the tuning of the Web container as opposed to the proxy.

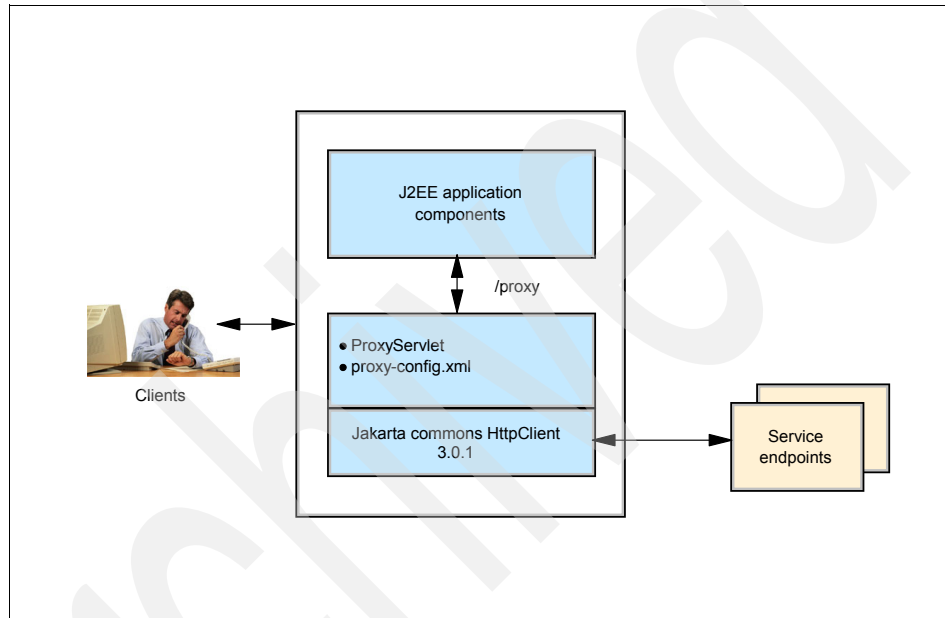


Figure 6-22 Enterprise Application Architecture after applying Ajax Proxy

A difference between the Ajax proxy and other WebSphere Application Server proxies such as the Web messaging channels is that the Ajax proxy is a reverse proxy, meaning that it is an application-level proxy. It must be embedded within an application.

The proxy-config.xml file can be modified using a text editor and must be in a location on the classpath accessible by the proxy servlet.

The proxy-config.xml file defines the policy by which URI requests are allowed to pass through the proxy and how context paths from the client will be mapped to the URI on a server. Changes to the proxy-config.xml file are not dynamic. Therefore, the servlet needs to be restarted for the changes to be recognized.

Ajax proxy can be packaged in two ways:

- ▶ Embedded as part of a Web application

As a servlet (`com.ibm.ws.ajaxproxy.servlet.ProxyServlet`) as part of the `AjaxProxy.jar` file, the proxy can be embedded in a J2EE application and deployed with the application. Embedding the proxy with the application allows the proxy to be deployed with the application in a ready-to-run configuration.

If you are building an application that combines the content from one or more service endpoints in an Ajax-based application you might choose to embed the Ajax proxy in your application. Modify your `application.xml` file or `geronimo-application.xml` (if using WebSphere Application Server Community Edition 2.0) to include the Ajax proxy Servlet. Normally, if you already have an enterprise EAR file created, you can associate the Ajax proxy for IBM WebSphere WAR file with your EAR project and Rational Application Developer will take care of the rest.

Modify the `proxy-config.xml` to define the URI context paths, URLs, and policies that the proxy will support. Build your application with the Ajax proxy servlet and deploy it to either WebSphere Application Server V6.x or WebSphere Application Server Community Edition 2.0.

- ▶ As a standalone Web application

As a servlet, the proxy can run as a standard Web application. Other applications can then use the proxy to broker requests. To do this, modify the `proxy-config.xml` to define URI context paths and policies that the proxy will support. Then deploy the Ajax proxy for IBM WebSphere WAR file to your application server.

Ajax proxies allow Web browsers to bypass the same-origin policy and therefore to access vendor servers using `XMLHttpRequest`. To bypass this the same-origin policy, you can choose from two approaches:

- ▶ The first approach is that the client-side Web application is aware of the vendor URL and passes it as a request parameter in the HTTP request to the Ajax proxy. The proxy then forwards the request to some `www.remoteservice.com`. Note that you can hide the use of a proxy server in the implementation of the Ajax library used by the Web application developer. From the Web application developer's point of view, it might appear that there is no same-origin policy at all.
- ▶ The second approach is that the client-side Web application is not aware of the vendor URL, and it tries to access resources on the Ajax proxy server through HTTP. By a predefined encoding rule, the Ajax proxy translates the requested URL into a vendor server URL and retrieves contents on behalf of

the client. In this case, it appears to the Web application developer that it is communicating with the proxy server directly.

### 6.3.4 Ajax proxy sample standalone implementation

In this section we show how to create a sample Ajax proxy application with the help of Rational Application Developer, configure it, and then deploy it on WebSphere Application Server.

1. Start Rational Application Developer in a new workspace and then close the Welcome window.
2. Create a new project by selecting **File** → **New** → **Project**. Select **Ajax Proxy** under IBM WebSphere Application Server Feature Pack for Web 2.0 and then click **Next**, as shown in Figure 6-23. Note that you may be required to select the Show All Wizards check box.

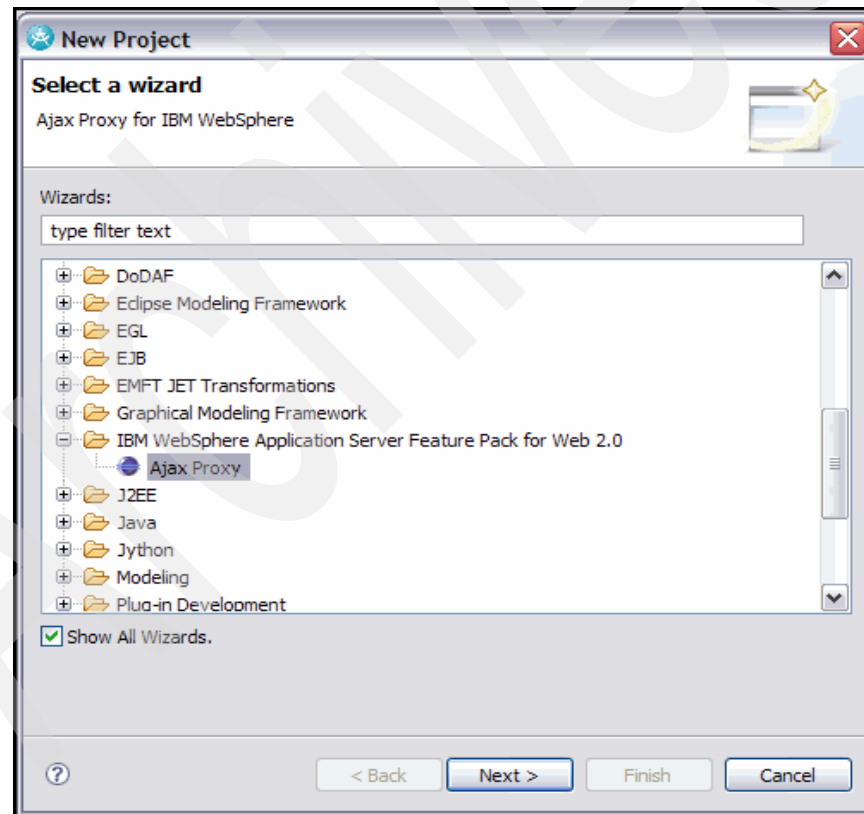


Figure 6-23 New Project wizard

3. Enter the name of the Web project as **ItsoAjaxProxy**, select the target runtime **WebSphere Application Server v6.1**, and check **Add project to an EAR**, as shown in Figure 6-24.

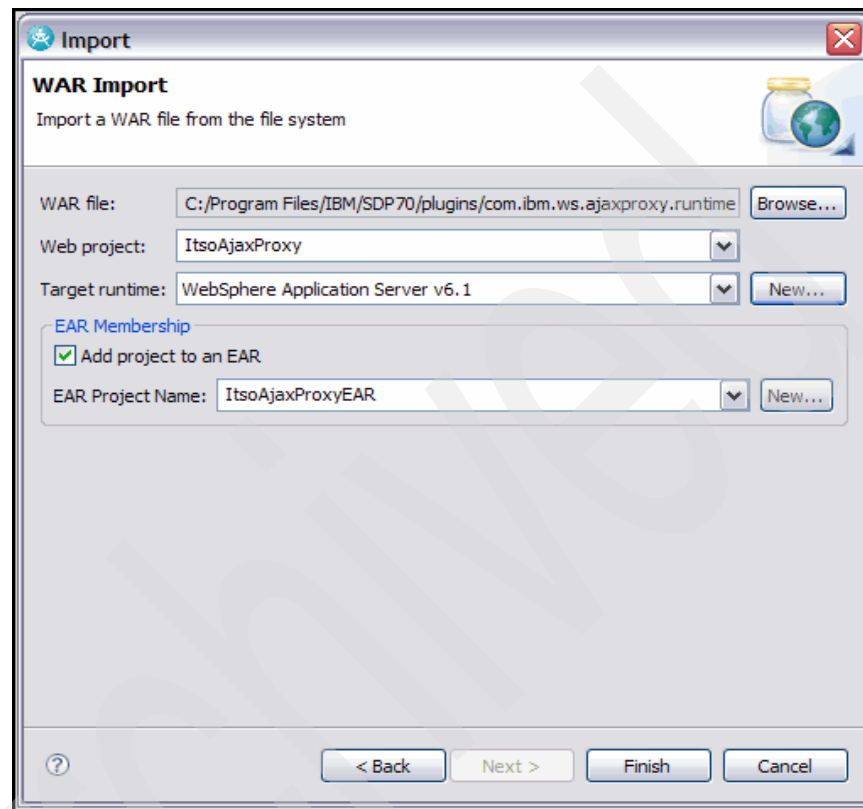


Figure 6-24 Import Web Archive dialog

4. Click **Finish** and switch to the Web perspective.

5. The Navigator and Project Explorer views show the complete project structure, which includes the AjaxProxy.jar file, the JSON4J.jar file, and additional jar library files in the lib folder, proxy-config.xml configuration file, or geronimo-Web.xml for WebSphere community edition server (Figure 6-25).

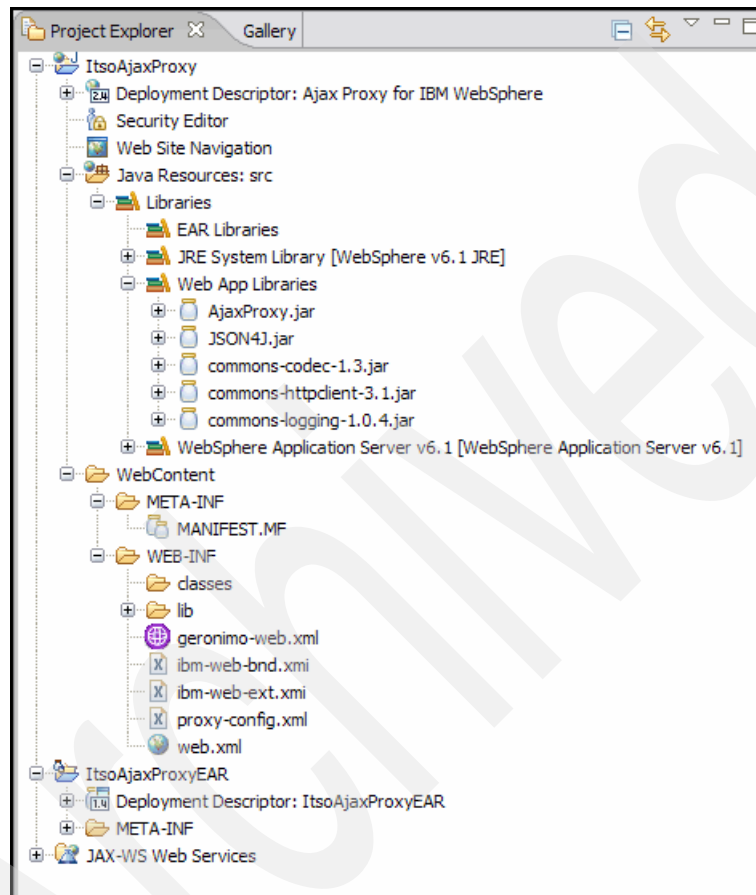


Figure 6-25 Ajax Proxy project structure

Notice the servlet-mapping for the ProxyServlet (com.ibm.ws.ajaxproxy.servlet.ProxyServlet) to URL /proxy/\* in the Web deployment descriptor Web.xml, as shown in Figure 6-26.

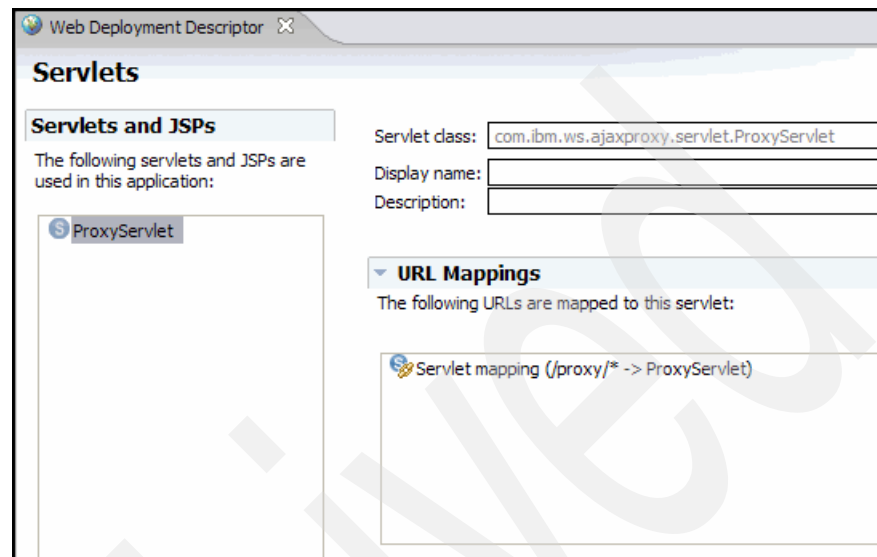


Figure 6-26 Proxy Servlet configuration in Web Deployment Descriptor

6. Open the proxy-config.xml file and you should see the proxy mapping parameters in the proxy rules section similar to those shown in Example 6-16.

*Example 6-16 proxy-config.xml file*

```
<proxy-rules
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

  xmlns:proxy="http://www.ibm.com/xmlns/prod/websphere/portal/v6.0.1/proxy-config">
  <proxy:mapping contextpath="/http/*" />
  <proxy:mapping contextpath="/rss/money_latest.rss"
url="http://rss.cnn.com" ></proxy:mapping>
  <proxy:mapping contextpath="/intl/*" url="http://www.google.com"
></proxy:mapping>
  <proxy:mapping contextpath="/redbooks" url="http://www.ibm.com"
></proxy:mapping>
  <proxy:mapping contextpath="/ibm" url="http://www.ibm.com"
></proxy:mapping>
  <proxy:mapping contextpath="/developerworks" url="http://www.ibm.com"
></proxy:mapping>
  <proxy:policy url="*" acf="none">
```

```
<proxy:actions>
  <proxy:method>GET</proxy:method>
</proxy:actions>
<proxy:cookies>
  <proxy:cookie>ABCD</proxy:cookie>
</proxy:cookies>
</proxy:policy>
</proxy-rules>
```

---

7. Start WebSphere Application Server from the Servers view.
8. The server status will change to *started*. Right-click **ItsoAjaxProxy Web project** in Project Explorer and select **Run As** → **Run on Server**. This publishes the application to the WebSphere Application Server.
9. Open a browser and set the address bar URL in the browser to `http://localhost:9080/ItsoAjaxProxy/proxy/http/www.ibm.com`.
10. This should display the IBM home page `ibm.com` because of the proxy mapping `"/http/*"` defined in the `proxy-config.xml` file.
11. Remove the line `<proxy:mapping contextpath="/http/*" />` from the `proxy-config.xml` file and then save it. Repeat the previous steps to publish and execute. This time you should receive an error message for not having an appropriate `contextpath` value defined.
12. Now add additional proxy mapping elements in the `proxy-config.xml` file as shown in Example 6-17 and again repeat the previous steps to publish and execute.

*Example 6-17 Adding more proxy mapping elements*

---

```
<proxy:mapping contextpath="/redbooks" url="http://www.ibm.com" />
<proxy:mapping contextpath="/developerworks" url="http://www.ibm.com" />
```

---

13. Set the address bar in your browser to:

`http://localhost:9080/ItsoAjaxProxy/proxy/redbooks`

It should open to the home page of IBM Redbooks:

`www.ibm.com/redbooks`

The routing of the URL request containing *redbooks* to the URL `www.ibm.com/redbooks` is accomplished by proxy mapping in the `profile-config.xml` file.

Similarly, you can try accessing *developerworks* instead of *redbooks*.



14. Set the address bar in your browser to:

<http://localhost:9080/ItsoAjaxProxy/proxy/developerworks/rational>

You should receive a CWXJX0501E error message, as shown in Figure 6-27



Figure 6-27 Invalid request error

15. Modify the proxy mapping within the proxy-config.xml file:

```
<proxy:mapping contextpath="/developerworks"
url="http://www.ibm.com" /> in proxy-config.xml to <proxy:mapping
contextpath="/developerworks/*" url="http://www.ibm.com" />
```

16. Again publish the Web project to the WebSphere Application Server.

17. Set the address bar in your browser to:

<http://localhost:9080/ItsoAjaxProxy/proxy/developerworks/rational>

You should now be able to see the Web page:

[www.ibm.com/developerworks/rational](http://www.ibm.com/developerworks/rational)

This is how the Ajax proxy server applies restrictions to URLs being accessed.

### 6.3.5 XML Schema of profile-config.xml

A complete XML Schema Definition for profile-config.xml is available in Appendix A, "XML schema definitions" on page 645.

### 6.3.6 Ajax Proxy limitations

In this section we discuss Ajax Proxy limitations.

#### HTML rewrites

The Ajax proxy does not attempt to rewrite the URL request in the body of the data returned in the HTTP response. As an example, the following proxy mapping rule `<proxy:mapping contextpath="/http/*" />` URL requests to the proxy with the context path `/http/www.myothersite.com` will redirect to `http://www.myothersite.com` and return the result to the browser. Consider the

image tag `<IMG href="/images/mypicture.jpg" />` file in an HTML page returned by `www.myothersite.com`. The browser issues a request for `/images/mypicture.jpg` to the proxy servlet, which responds with a 404 error. The proxy does not resolve that the image request is intended for `www.myothersite.com`.

## Security

The proxy does not provide a filtering IP address. As an example, even though the proxy is configured to be used by your application, another application can access the proxy. If you need to secure the proxy, then complete this action in the context of J2EE security. The WebSphere Application Server information center provides additional information about how to secure your J2EE application.

## Java libraries

This chapter describes how to use two powerful Java libraries that are included in the Web 2.0 Feature Pack toolkit. These libraries cover two key areas of Web 2.0 application construction. The first enables your WebSphere application to produce and consume data in JavaScript Object Notation (JSON) format.

The second library, called FeedSphere, offers a rich programming interface to the Apache Abdera feed libraries that support the creation, management, and usage of Feed-based applications.

This chapter explains how to set up these libraries within your development project and how you can quickly start using them to build Web 2.0 solutions using the feature pack.

This chapter covers the following Java Libraries:

- ▶ JavaScript Object Notation for Java Library (JSON4J)
  - Java Object to JSON
  - JSON to Java Object
- ▶ FeedSphere (Abdera Libraries)
  - Atom Publish
  - Atom

## 7.1 Java libraries

The Java libraries that are included in the Web 2.0 Feature Pack are designed to assist the developer in accessing data from an enterprise system and presenting it in a form that can be consumed by Web 2.0 applications. The two most significant formats that are building the foundation of interoperability are JavaScript Object Notation (JSON) and Atom and Real Simple Syndication (RSS) feed syndication formats.

If you want to create a Rich Internet application with Dojo or are required to deliver dynamic feeds of information for consumption into a mashup, then these libraries will be the key to enabling you to create the content quickly.

This chapter explains some of the background of the formats and why they were created. It focuses on presenting real hands-on examples using the Java libraries.

## 7.2 Overview

Why do we need new data formats if Web Services using SOAP was all that was ever needed? This is a common question that is asked when you start to look at the reasons why Web 2.0 is becoming so pervasive in the next generation of Internet applications.

The answer is quite simple. The primary reason that SOAP and other XML dialects have been so hard to consume by Rich Internet application developers is because of the lack of native support within the browsers. It is relatively simple to parse an XML string, but this puts extra processing time on the client application. Imagine a situation where a client application has a large XML structure and needs to navigate its Document Object Model (DOM) to retrieve a single string value that will then be placed into the browser page DOM. This will require a significant amount of time, code, and effort to perform this operation for each element of data.

Over the last few years it has been very common for Internet Web sites to release content in a syndication format. Most of the popular Web browsers now have built-in support for RSS and Atom syndication format specification feeds. It is very easy to keep track of new articles and update new articles. RSS and Atom can be used for a wide variety of application types other than just news feeds. They can be just as useful with business data or dynamic data. The next generation of Web application is going to allow feeds to be produced and consumed in entirely new and innovative ways. Taking customer data and mixing it with sales and linking it to a map application can produce dynamic mashups.

## JSON4J

To support JSON creation and consumption IBM has developed the JSON4J Java library. It has been designed to allow Java developers to easily create JSON from Plain Old Java Objects (POJOs). This library can be easily installed as a shared library into the application server run time or can be included within an WebSphere application resource project.

### FeedSphere (Apache) Abdera libraries

Abdera is an incubator project within the Apache organisation. Its goal is to build a functionally complete, high-performance implementation of the IETF Atom Syndication Format (RFC 4287) and Atom Publishing Protocol specifications.

**Note:** More information about the syndication specification can be found at:

<http://www.ietf.org/rfc/rfc4287.txt>

IBM has included this library within the Web 2.0 feature pack to enable Java developers to produce and manage Atom-based syndication streams using a comprehensive Java library.

**Note:** More information about Abdera can be found at:

<http://incubator.apache.org/abdera/>

## 7.3 JSON4J

This section describes in detail how the JSON4J Java Library can be used within a WebSphere application.

More information about JSON can be found at:

<http://www.json.org>

### 7.3.1 Introduction

JavaScript Object Notation is a subset of the JavaScript language, and for this reason JSON streams can be turned into JavaScript objects by using the eval statement within the language.

JSON is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language, Standard ECMA-262 3rd Edition -

December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

- ▶ A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- ▶ An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

JSON is based upon the JavaScript language. It can be evaluated into a JavaScript object structure and then be used within the JavaScript language. This makes it ideal for Web 2.0 Rich Internet application development with JavaScript libraries like the Dojo Toolkit.

## Languages supported

What makes JSON so popular is its simple and easy-to-read format and the availability of a wide variety of languages that support it. The syntax is language independent and uses a set of constructs similar to those used within most modern programming languages.

This makes the task of building heterogeneous Web 2.0 solutions quite easy. JSON-formatted data can be generated by any of the leading application-serving run times on the market today.

*Table 7-1 Languages that support JSON*

Languages supporting JSON		
ActionScript	Java	Python
C	JavaScript	Ruby
C++	Lisp	Squeak
C#	Objective-C	E
Delphi	Perl	PHP

The language is text based on which will enable it to be generated by most popular languages and allow it to transfer using IP-based protocols. Delivering

JSON format structures over HTTP will be a common pattern for using JSON in Web 2.0 style applications.

The language is very lightweight in terms of syntax and complexity. This has enabled a number of parsers to be developed for a wide selection of programming language. This fact will make JSON a common choice for application integration.

A final comment is that JSON is not a document format. It was designed for data exchange only and for this reason additional standard formats with a fixed structure and syntax may develop based on JSON. JSON-RPC is one area where this has been standardized.

## 7.3.2 JSON format

In this section we discuss the JSON format.

### Object

In JSON an object has these forms. An object is an unordered set of name/value pairs. An object begins with a left brace ( { ) and ends with a right brace ( } ). Each name is followed by a colon ( : ) and the name/value pairs are separated by a comma ( , ).

This structure can be used to create content that represents objects, as shown in Example 7-1.

*Example 7-1 Sample JSON object structure*

---

```
{
  "firstname" : "Matthew",
  "surname" : "Perrins",
  "age" : 40
}
```

---

### Array

An array is an ordered collection of values. An array begins with a left bracket ( [ ) and ends with a right bracket ( ] ). Values are separated by a comma ( , ).

JSON objects can be grouped into lists of contents for an array. The simple object structure and its ability to support arrays mean that it is ideal for handling Java POJOs, as shown in Example 7-2.

*Example 7-2 JSON array example*

---

```
[
  {"firstname":"Matthew","surname":"Perrins","age":40},
  {"firstname":"Tania","surname":"Perrins","age":33},
  {"firstname":"Fintan","surname":"Perrins","age":6},
  {"firstname":"Joe","surname":"Perrins","age":4},
  {"firstname":"Tabitha","surname":"Perrins","age":2}
]
```

---

An array of structures can contain a variety of object typeObject structure arrays can be used at multiple levels. You can see in Example 7-3 that the customer value is another object and that the orders value is made up of an array of order objects.

*Example 7-3 Example of a nested array of JSON objects*

---

```
{
  "customer" : { "name":"ABC Inc","number":"123456" },
  "orders" :
  [
    { "number":"89374","product":"XYZ-29384",":qty":5},
    { "number":"26543","product":"XYZ-11223",":qty":8},
    { "number":"67561","product":"XYZ-99876",":qty":12}
  ]
}
```

---

## Value

A value can be a string in double quotation marks, a number, a true or false or null, or an object or an array. These structures can also be nested.

## Strings

A string is a collection of zero or more Unicode characters, wrapped in double quotation marks, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string.

A number is represented very much like a C or Java number, except that the octal and hexadecimal formats are not used.

Whitespace can be inserted between any pair of tokens. Except for a few encoding details, that completely describes the language.



## JSON is not XML

It is important to remember that JSON is not an alternative to Extensible Markup Language (XML). XML is a set of syntax rules used to define special purpose languages that meet the diverse needs of business, science, and government, as well as the publishing world.

The term XML also refers more broadly to the set of related technical standards created with XML syntax, as well as to custom (non-standard) languages based on these standard specifications.

Collectively, the family of XML specifications enables the interchange of data and structured text across dissimilar computer systems, especially when the Internet is the transmission medium.

Table 7-2 shows that JSON is a good choice as a data exchange mechanism between servers, programming languages, and as the core data format for Web 2.0 Rich Internet applications. If you require more comprehensive support for schemas, namespaces, and other capabilities then continue to use XML as your data exchange mechanism.

*Table 7-2 JSON comparison with XML*

JSON	XML
objects	element
arrays	attribute
strings	attribute string
numbers	content
booleans	<![CDATA[]]>
null	entities
	declarations
	schema
	stylesheets
	comments
	version
	namespace

## Plain Old XML (POX)

Within the Web 2.0 world the use of POX is commonplace. POX basically makes use of the XML format for simple data layout and containment. Some of the computer-generated XML can be very complex to read and consume within the browser. The use of POX allows for applications on the server to still generate XML, but in a format that is easy to consume. The reason for using POX may be that you have a client to your Web 2.0 service that has the mechanisms for processing, creating, and manipulating XML.

You may have existing service clients that are already configured to process XML as a data format. For future interaction with the Model View Controller elements of a Rich Internet application you will find JSON far more convenient. XML can be a little cumbersome in JavaScript to navigate the elements of data that you need.

Example 7-4 shows the basic structure of POX as XML always intended (that is, the clean usage of tags with few overriding of attributes).

*Example 7-4 An example of POX*

---

```
<person>
  <firstname>Matthew</firstname>
  <surname>Perrins</surname>
  <age>40</age>
</person>
```

---

### 7.3.3 Consuming JSON in a browser

JSON is based on the JavaScript Object Notation, and because of that it is very easy to consume by JavaScript within the browsers in common use. You can easily create the JSON objects and then consume them in programs. The Dojo Toolkit has added support for serializing and deserializing JSON structures. The new ECMA JavaScript standard intends to support these mechanism natively in the language. If you are looking to create Dynamic Ajax-based applications then you should consider using JSON as your service data payload and then consuming it within your browser.

Example 7-5 shows how simple and easy to read JSON structures are. This example shows an object with name value pairs being used to represent the first name, surname, and age of a person.

*Example 7-5 JSON structure*

---

```
{
  "firstname" : "Matthew",
  "surname" : "Perrins",
  "age" : 40
}
```

---

The following code snippet in Example 7-6 shows that any string that contains a JSON structure can be evaluated and then used within the JavaScript language.

*Example 7-6 Example of JSON being used in JavaScript*

---

```
// Sample Person
var strper = '{ "firstname":"Matthew","surname":"Perrins","age":40}';
var person = eval("(" + strper + ")");
document.myform.firstname.value = person.firstname;
document.myform.surname.value = person.surname;
document.myform.age.value = person.age;
```

---

Observe that the JSON string structure is first evaluated into a JavaScript object. This object can then be accessed using standard JavaScript mechanisms. This includes accessing field variables and arrays.

### 7.3.4 Create a project

To start using the JSON4J library you need to decide whether to place the Java Library within the Dynamic Web project or within the WebSphere Application Server run time.

The library can be configured in two ways to be used with the development environment:

- ▶ The first is by including the JSON4J.JAR within the WEB-INF/lib directory.
- ▶ The second is by configuring a shared library within the WebSphere Application Server run time.

The following sections explain how this can be achieved. The first step is to create a dynamic Web project within the Rational Application Developer tooling:

1. Open the Rational Application Developer tooling with a predefined Workspace that will allow you to test these scenarios.
2. Select **File** → **New** → **Project**. This opens the project selection dialog.
3. Expand the Web wizard folder and select **Dynamic Web Project** → **Next**, as shown in Figure 7-1. The New Dynamic Web Project wizard page now appears.

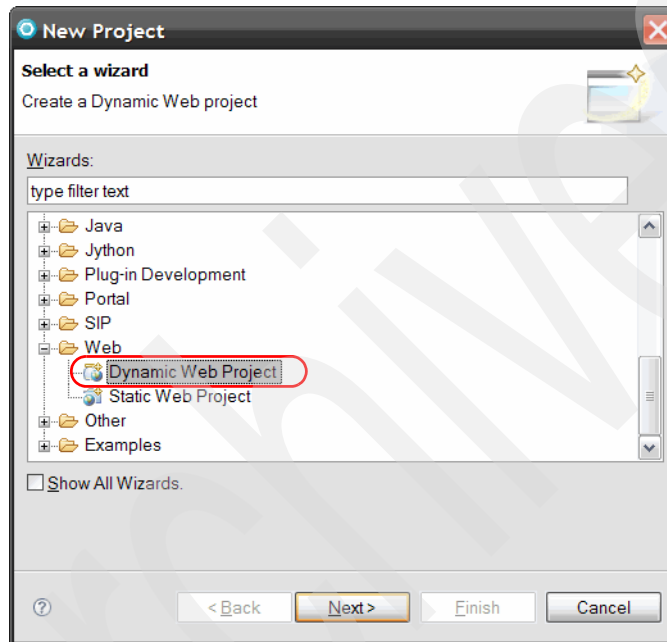


Figure 7-1 File → New → Project dialog

4. For the project name enter ITS0JS0N and check that the EAR project name that is auto-completed matches the project name.
5. Select a target run time that has the Web 2.0 Feature Pack installed within it.

6. Select **Next** to move to the next page of the wizard. See Figure 7-2.

The screenshot shows the 'New Dynamic Web Project' dialog box. The 'Project name' field contains 'ITSOJSON'. The 'Project contents' section has 'Use default' checked, and the 'Directory' is 'D:\Workspace\ITSO\ITSOJSON'. The 'Target Runtime' is set to 'WebSphere Application Server v6.1 Web 2.0 FEP'. The 'Configurations' section shows '<custom>'. The 'EAR Membership' section has 'Add project to an EAR' checked, and the 'EAR Project Name' is 'ITSOJSONEAR'. The 'Next' button is highlighted in yellow.

Figure 7-2 Specify name for dynamic Web project

7. Leave the project facets list set to its defaults, and then click **Next**. See Figure 7-3.

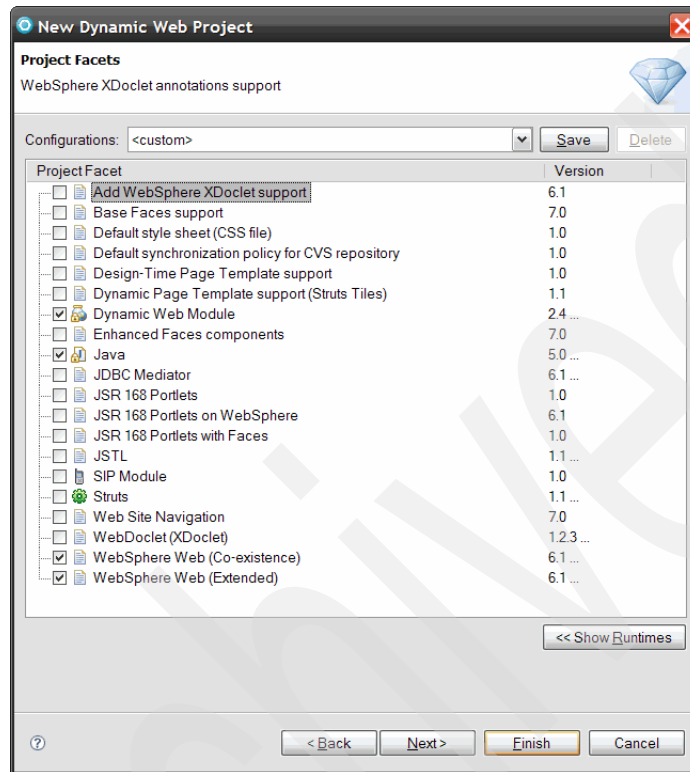


Figure 7-3 Dynamic Web project facets select

8. Unless you need to change the context root of the application, leave the Web Module configuration page set to its defaults. Select **Finish**, as shown in Figure 7-4.

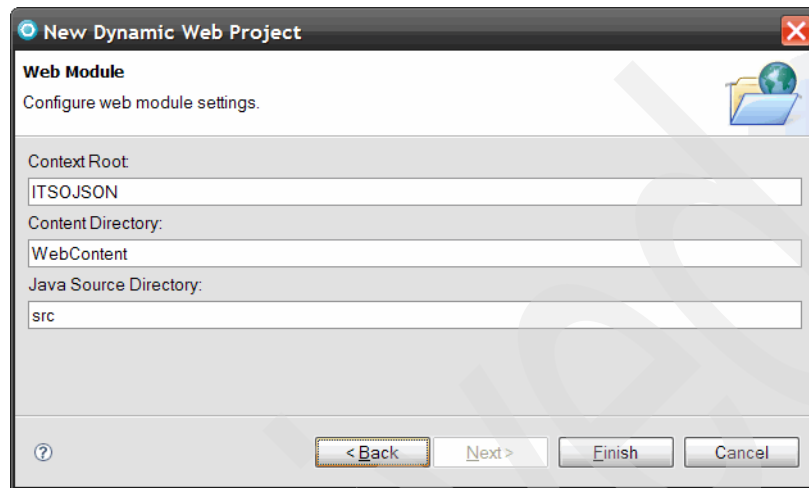


Figure 7-4 Web Module configuration

Figure 7-5 shows your project within the Project Explorer view of the Rational Application Developer workbench.

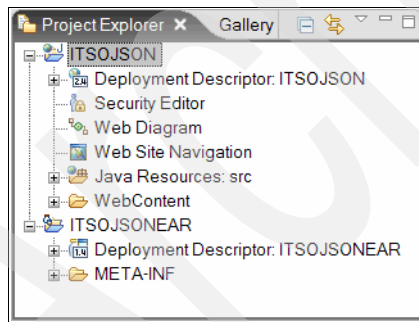


Figure 7-5 Project Explorer after the dynamic Web project has been created

You are now ready to configure the JSON4J.JAR for inclusion within your project.

### 7.3.5 Web project configuration

A common method for including the JSON4J.JAR within your project is to include it within the Web Project lib directory. It will then be included within the Enterprise Application Resource (EAR) file when the application is deployed. Packaging the

JSON4J.JAR file with the EAR file means that the server will not require any configuration changes in order to host the Java Libraries.

**Best practice:** If you are planning on deploying multiple applications that will all use the JSON4J libraries then it is a best practice to configure the library as a shared resource and allow each Web Application access to it at run time. See section XXXX for details on how to do this.

To configure the Web project:

1. To include the JSON4J.JAR file in your Dynamic Web Application select **File → Import → File System**.
2. Use the **Browse** button for the from directory to navigate and select the directory where the JSON4J.JAR file is located after the installation of the feature pack. This is usually below the base WebSphere installation directory.
3. Once the directory is displayed, select the **JSON4J.JAR** file, as shown in Figure 7-6.

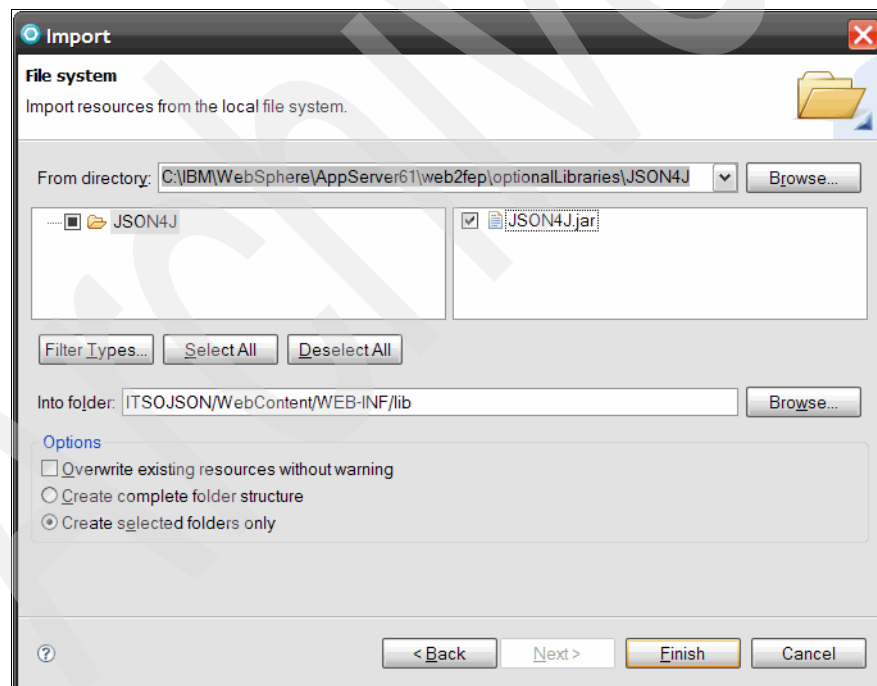


Figure 7-6 Import the JSON4J.JAR file into the Web project

4. In the Into folder navigate and select the **WebProject/WEB-INF/LIB** directory.



5. You can check whether the jar has been imported by expanding the folder named **Web App Libraries** underneath the Java Resources folder for the project. See Figure 7-7.

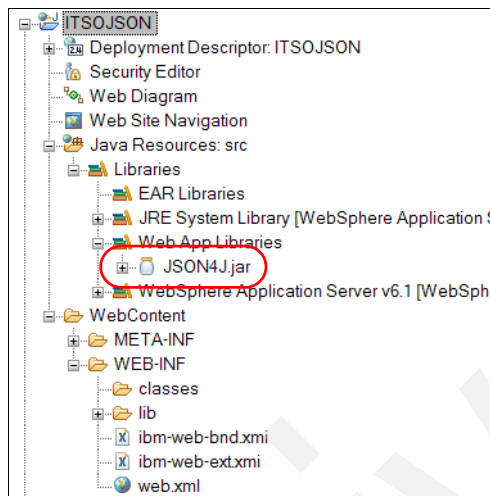


Figure 7-7 Check whether library is included in Project Explorer

You are now ready to start using the library within your application code.

### 7.3.6 Configuring WebSphere shared libraries

Shared libraries contain the files used by multiple applications. Each shared library consists of a symbolic name, a Java class path, and a native path for loading Java Native Interface (JNI™) libraries. You can use shared libraries to reduce the number of duplicate library files on your system.

You use the administrative console to define a shared library. You can define a shared library at the cell, node, or server level.

**Note:** On a Version 6.1.0.11 and later multiple-server product, you also can define a shared library at the cluster level.

A separate class loader is used for shared libraries that are associated with an application server. This class loader is the parent of the application class loader, and the WebSphere Application Server extensions class loader is its parent. Shared libraries that are associated with an application are loaded by the application class loader.

Defining a library at one of the these levels does not automatically place the library into the class loader for an application server. You must associate the library to an application or server before the classes represented by the shared library are loaded into either a server-wide or application-specific class loader.

Using the administrative console, you can then associate the library with an application, module, or server to load the classes represented by the shared library in either a server-wide or application-specific class loader.

If your deployed applications use shared library files, define shared libraries for the library files and associate the libraries with specific applications or with an application server. Associating a shared library file with a server associates the file with all applications on the server. Use the administrative console Shared Libraries page to define new shared library files to the system and to remove existing library files from the system.

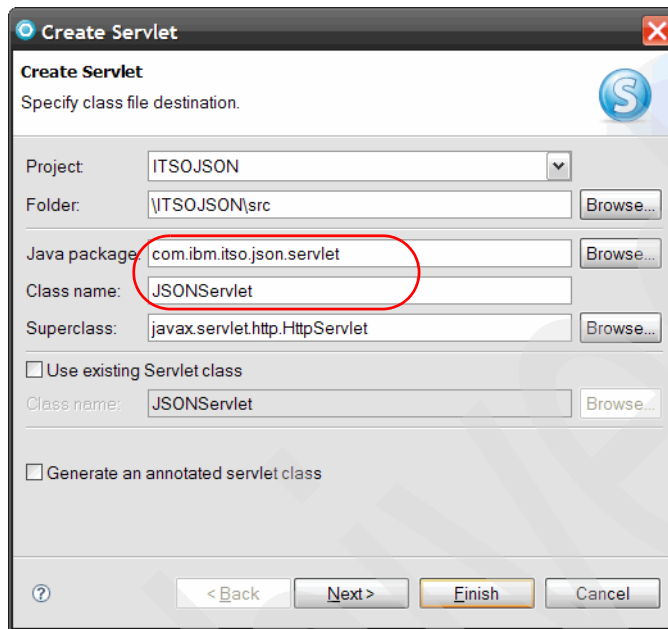
### 7.3.7 Sample invocation servlet

To demonstrate the use of the JSON library we will create a simple servlet that controls the creation of a set of samples to show how the JSON library can be used within an existing dynamic Web project. This servlet will include code to show how POJOs can be converted directly into a JSON-formatted string.

To create the servlet:

1. Using the Dynamic Web project already created in the previous sections, open the Deployment Descriptor editor from the project navigator.
2. Click the **Servlet** tab at the bottom to bring you to the Servlets view.
3. Click the **Add** button to add a servlet to the project.

4. Enter the Java package of `com.ibm.itso.json.servlet` and the servlet name of `JSONServlet`, then click **Next**, as shown in Figure 7-8.



The image shows a 'Create Servlet' dialog box with the following fields and options:

- Project:** ITSOJSON
- Folder:** \ITSOJSON\src
- Java package:** com.ibm.itso.json.servlet (highlighted with a red circle)
- Class name:** JSONServlet (highlighted with a red circle)
- Superclass:** javax.servlet.http.HttpServlet
- ☐ Use existing Servlet class
- ☐ Generate an annotated servlet class

At the bottom, there are buttons for '< Back', 'Next >', 'Finish', and 'Cancel'. The 'Next >' button is highlighted.

Figure 7-8 Add a servlet to help with testing of the JSON Library

5. Select the default URL mappings that have been generated and then select the **Edit** button, as shown in Figure 7-9.

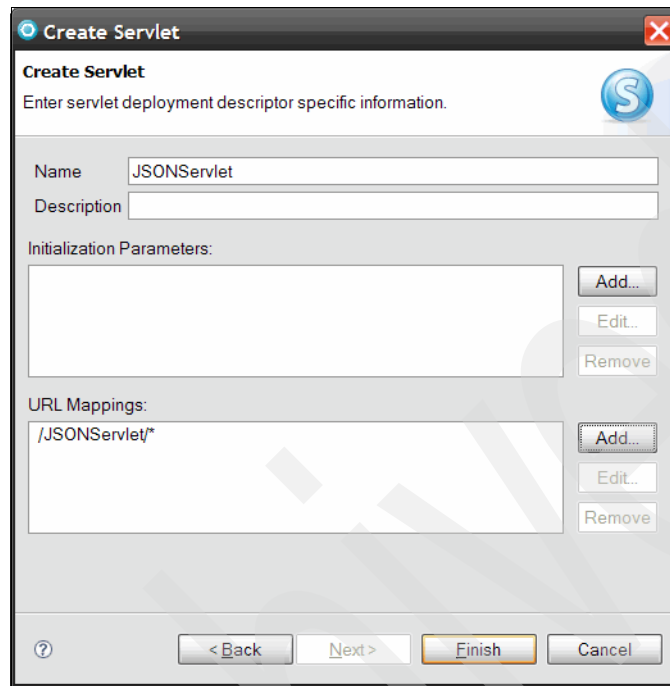


Figure 7-9 URL mappings for servlet

6. Edit the Mapping to be /JSONServlet/\* and select **OK**, as shown in Figure 7-10.

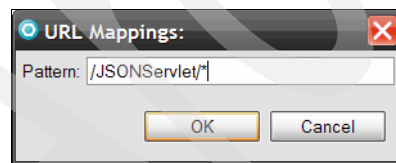


Figure 7-10 Servlet URL Mapping

7. Check that the URL mapping for the servlet is defined as /JSONServlet/\* and then select **Finish** to complete the creation of the servlet.

8. After creating the servlet use the code shown in Example 7-7 to build out the skeleton.

*Example 7-7 Servlet skeleton code*

---

```
package com.ibm.itso.json.servlet;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.HashMap;
import java.util.Map;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class JSONServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;

    public void process(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {

        String contentType = req.getContentType();
        String path = req.getPathInfo();

        res.setContentType("text/html");
        res.setStatus(HttpServletResponse.SC_OK);

        /***** CODE TO BE INSERTED HERE *****/
    }

    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        process(req, resp);
    }

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {

        process(req, resp);
    }
}
```

```
}  
  
}
```

---

This servlet shell will now be used in the following samples.

9. Save the servlet and the deployment descriptor for the project.
10. We will now create a simple test harness to launch the tests. The simplest method is to create an HTML file using A HREF tags to invoke the URLs that we want to test.
11. Create a blank HTML file called index.html in the Web Content directory.
12. Add the contents shown in Example 7-8 to the file and then save it.

*Example 7-8 Contents to add to file*

---

```
<html>  
<title>JSON Samples</title>  
<body>  
<h3>JSON - Samples</h3>  
<ol>  
<li><a href="JSONServlet/object">Object</a></li>  
<li><a href="JSONServlet/array">Array</a></li>  
<li><a href="JSONServlet/pojo">POJO</a></li>  
</body>  
</html>
```

---

You can see from the A HREF definitions that the servlet will use the path attribute to map each request to a specific test case.

### 7.3.8 JSON object and JSON array

The core JSON4J library utilizes two principle objects for the JSON structure. They are:

- ▶ **JSONObject** class, which holds the data for an object using a map-style interface using value name pairs
- ▶ **JSONArray** class, which holds a set of JSONObject instances in a collection-style interface

To demonstrate the use of the JSON object add the snippet of code shown in Example 7-9 to your test servlet. This code creates an instance of the JSON object and then populate it with sample data. It will then serialize the object into a JSON formatted string.

*Example 7-9 Creating a basic JSON object*

```
if (path.equals("/object")) {  
    JSONObject obj = new JSONObject();  
    obj.put("firstname", "Joe");  
    obj.put("surname", "perrins");  
    obj.put("age", new Integer(4));  
  
    String jsonStr = obj.serialize(true);  
    PrintWriter writer = res.getWriter();  
    writer.print(jsonStr);  
    writer.flush();  
}
```

You can see from this example that the JSON object contains name value pairs being supplied as a JSON object. The JSON object can then be serialized into a string type that is then returned as the response. Note that this sample code is first mapped to the /object path on the servlet.

To test the servlet add the Web project to an instance of the WebSphere Application Server test environment and run the application. You should see the Web page in the embedded browser, as shown in Figure 7-11.



*Figure 7-11 JSON Samples test page*

Click the **Object** link to test the code snippet. The results are shown in Figure 7-12.



Figure 7-12 Object result

Figure 7-12 shows the JSON object after it was serialized and converted into a compliant JSON string that can now be used within the browser.

### 7.3.9 Using JSON within a browser

This section describes how you can create a raw Ajax client running within a browser page and integrate it with the simple JSON object example explained in the previous section.

**Note:** What do we mean by raw Ajax? This term refers to the use of the basic XMLHttpRequest support within the browser and the JavaScript language necessary to invoke it. Most modern applications would use a framework like Dojo to perform asynchronous calls to a WebSphere server.

The purpose of this sample is to retrieve an object in JSON format from the server and then to display it inside the HTML form.

To build a raw Ajax client and integrate it with the simple JSON Object example:

1. Within the existing dynamic Web project create a new HTML file called `ajaxsample.html`.
2. Add the code snippet shown in Example 7-10 to the `ajaxsample.html` file.

*Example 7-10 Ajax sample HTML snippet*

```
<title>Ajax JSON Sample</title>
<body>
<h3>Person</h3>
<p></p>
<form name="myform" action="#">
<table>
```



```

<tr>
<td><a href="#" onclick="read()">Read</a>
<td><a href="#" onclick="clearForm()">Clear</a>
</tr>
<tr>
<td>Name</td>
<td><input type="text" name="firstname" size="20"></td>
</tr>
<tr>
<td>Surname</td>
<td><input type="text" name="surname" size="20"></td>
</tr>
<tr>
<td>Age</td>
<td><input type="text" name="age" size="20"></td>
</tr>
</table>
</form>
<input type="submit" name="Submit" value="Submit">
</body>
</html>

```

---

3. At the top of the project create a `<script>` tag section, as shown in Example 7-11.

*Example 7-11 Script tag section*

```

<script>
</script>

```

---

4. Declare a function that will retrieve a valid `XMLHttpRequest` object from within a Firefox or Internet Explorer browser by adding the function shown in Example 7-12.

*Example 7-12 Retrieving a valid `XMLHttpRequest` object*

```

var xmlhttp;

function createXMLHttpRequest() {

    if(window.ActiveXObject) {
        xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
    }
    else {

```

```

        xmlHttp = new XMLHttpRequest();
    }
}

```

---

5. The next step is to implement the read() method. This invokes an asynchronous call to the servlet previously created. Add the code shown in Example 7-13 to the script tag.

*Example 7-13 Ajax call to the server*

---

```

function read() {
    createXMLHttpRequest();
    xmlHttp.onreadystatechange = handleStateChange;
    xmlHttp.open("GET","/ITS0JSON/JSONServlet/object",true);
    xmlHttp.send(null);
}

```

---

6. A call to the read() function creates an instance of the XMLHttpRequest object and then constructs its parameters. Notice that the URL for the servlet is defined to match the sample that was previously created. Another important parameter is the state change method, which is a new method created to handle the response back from the server.
7. Add the code snippet shown in Example 7-14 to the script tag to handle the server response.

*Example 7-14 Handle state change function and code for updating the form*

---

```

function handleStateChange()
{
    if (xmlHttp.readyState == 4 ) {
        if (xmlHttp.status == 200) {
            var person = eval("(" + xmlHttp.responseText + ")");
            if(person != null)
                updateForm(person);
        }
    }
}

function updateForm(person)
{
    document.myform.firstname.value = person.firstname;
    document.myform.surname.value = person.surname;
    document.myform.age.value = person.age;
}

```

```
function clearForm()
{
    document.myform.reset();
}
```

---

8. You can see that the `handleStateChange` method checks that a valid response has been returned. It first checks the `xmlHttp.status` value to determine whether it matches a correct HTTP request. The function then converts the `xmlHttp.responseText` into a JavaScript variable using the `eval` function. This variable is then passed to the `updateForm` method to update the values of the form.
9. Link to this page from a Web browser and select the **Read** link, as shown in Figure 7-13. You should see that the values returned by the servlet are now placed in the form. The interesting part from a developer's point of view is that the data is converted into a JavaScript object directly by JavaScript and the member fields of the object can be directly accessed without the extra step of navigating and parsing the response. This enables you to quickly develop a powerful and rich application.



Figure 7-13 Ajax sample result page after a read has been performed

### 7.3.10 JSON arrays

The JSON4J library has the ability to handle arrays of objects. The JSON4J library includes a class named `JSONArray` that can contain many JSON objects. The following sample demonstrates how these arrays of JSON objects can be managed.

Using the dynamic Web project previously created, add the code shown in Example 7-15 to the JSONServlet.

*Example 7-15 Creating a JSON array object*

---

```
else if (path.equals("/array")) {

    JSONArray array = new JSONArray();

    JSONObject obj = new JSONObject();
    obj.put("firstname", "Fintan");
    obj.put("surname", "Perrins");
    obj.put("age", new Integer(6));
    array.add(obj);

    obj = new JSONObject();

    obj.put("firstname", "Tabitha");
    obj.put("surname", "Perrins");
    obj.put("age", new Integer(2));
    array.add(obj);

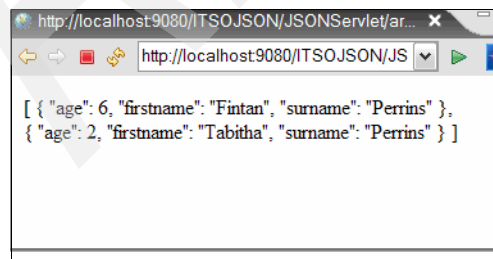
    String jsonStr = array.serialize(true);
    PrintWriter writer = res.getWriter();
    writer.print(jsonStr);
    writer.flush();

}
```

---

You can see that a JSON array object is first created and the JSON objects are added into it. Finally, the JSON array is serialized into a string that is returned in the response to the servlet request.

You should see the results shown in Figure 7-14 after running the sample with the JSON servlet/array URL.



*Figure 7-14 Array JSON example*

### 7.3.11 Java objects to JSON

Now that you have seen a basic demonstration of the JSON4J library functions, this section shows how more complex structures can be rendered into JSON format strings to be consumed by Rich Internet applications.

The JSON4J library has a limitation in that it only supports serialization of its own JSONObject and JSONArray objects. In order to integrate the JSON4J library into your existing JEE assets you will require a means that uses POJO introspection to automatically generate these objects based on the signature of the POJO.

The following sample performs this function and can be included in the servlet for testing.

The first step is to create a class that can process any given object and by using introspection turn it into a list of JSON objects that are contained within a JSON array. This class will need to work recursively to walk through the object hierarchy.

Under the src folder of the dynamic Web project create a class call POJOTOJSON.java and add the code shown in Example 7-16.

*Example 7-16 Sample POJOTOJSON class*

---

```
package com.ibm.itso.json.servlet;

import java.beans.BeanInfo;
import java.beans.IntrospectionException;
import java.beans.Introspector;
import java.beans.PropertyDescriptor;
import java.lang.reflect.Field;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import com.ibm.json.java.JSONArray;
import com.ibm.json.java.JSONObject;

public class POJOTOJSON {

    public static final String JREF_PREFIX = "$jref:";
```

```

private POJOToJSON() {}

public static Object serialize(Object bean) throws
IllegalArgumentException, IntrospectionException,
IllegalAccessException, InvocationTargetException {
    return(serialize_recursive(bean, new HashMap<Object, String>(),
JREF_PREFIX + "this"));
}

static Object serialize_recursive(Object bean, Map<Object, String>
objects, String jref) throws IllegalArgumentException,
IntrospectionException, IllegalAccessException,
InvocationTargetException {

    if(bean == null) {
        return(null);

    } else if((bean instanceof Boolean) || (bean instanceof Number)
|| (bean instanceof JSONObject) || (bean instanceof JSONArray)) {
        return(bean);

    } else if(bean.getClass().isPrimitive() || bean instanceof
String) {
        return(bean.toString());

    } else if(bean instanceof Map) {
        String matchedRef = (String) objects.get(bean);
        if(matchedRef != null) {
            return(matchedRef);
        }

        Map map = (Map) bean;
        JSONObject jo = new JSONObject();

        Iterator keyIter = map.keySet().iterator();
        while(keyIter.hasNext()) {
            Object key = keyIter.next();
            jo.put(key, serialize_recursive(map.get(key), objects, jref
+ "." + key));
        }

        return(jo);
    }
}

```

```

    } else if(!bean.getClass().isArray() && !(bean instanceof
Collection)) {
        String matchedRef = (String)objects.get(bean);
        if(matchedRef != null) {
            return(matchedRef);
        }

        objects.put(bean, jref);

        JSONObject jo = new JSONObject();
        BeanInfo bi = Introspector.getBeanInfo(bean.getClass(),
Object.class);

        Map<String, Field> publicFields = new HashMap<String,
Field>();
        Field fields[] = bean.getClass().getFields();

        for(int f = 0; f < fields.length; f++ )
        {
            Field field = fields[f];
            int mod = field.getModifiers();

            if(Modifier.isPublic(mod) && !Modifier.isFinal(mod)) {
                publicFields.put(field.getName(), field);
            }
        }

        PropertyDescriptor pds[] = bi.getPropertyDescriptors();
        for (int p = 0; p < pds.length; p++)
        {
            PropertyDescriptor pd = pds[p];

            // Try to use a getter method first.
            Method getter = pd.getReadMethod();
            Method setter = pd.getWriteMethod();
            if((getter != null) &&
Modifier.isPublic(getter.getModifiers())
                && ((setter != null) ||
publicFields.containsKey(pd.getName())) {
                Object value = getter.invoke(bean, (Object []) null);
                jo.put(pd.getName(), serialize_recursive(value, objects,
jref + "." + pd.getName()));
                publicFields.remove(pd.getName());
            }
        }
    }
}

```

```

        Iterator iter = publicFields.keySet().iterator();
        while(iter.hasNext()) {
            String fieldName = (String) iter.next();
            Field field = (Field) publicFields.get(fieldName);

            jo.put(fieldName, serialize_recursive(field.get(bean),
objects, jref + "." + fieldName));
        }

        return(jo);
    } else {
        Object[] entries;

        if (bean instanceof Collection) {
            entries = ((Collection) bean).toArray();
        } else {
            entries = (Object[]) bean;
        }

        JSONArray ja = new JSONArray();

        for (int i = 0; i < entries.length; i++) {
            Object beanElement = entries[i];
            ja.add(serialize_recursive(beanElement, objects, jref + "["
+ i + "]""));
        }

        return(ja);
    }
}

```

---

This POJOTOJSON class can now be used as a mechanism to serialize your POJOs into a format that will allow you to convert them into a JSON text stream.



Complete the POJOToJSON servlet by adding the writeJSONObject method to the servlet as shown in Example 7-17.

*Example 7-17 Servlet method for serializing a POJO into a JSON stream*

---

```
protected void writeJSONObject(HttpServletResponse res, Object[]
object)
    throws ServletException, IOException {

    PrintWriter writer = res.getWriter();

    Object json = null;
    try {
        json = POJOToJSON.serialize(object);
        if (json instanceof JSONObject) {
            ((JSONObject) json).serialize(writer, true);
        } else if (json instanceof JSONArray) {
            ((JSONArray) json).serialize(writer, true);
        } else {
            writer.print(json);
        }
    } catch (Exception e) {
        throw new IOException("JSON Serlize failed");
    }
}
```

---

You can see that the writeJSONObject method uses the POJOToJSON class to serialize the object and then uses the local serialization of the JSONObject and JSONArray classes to convert the content into a type string that can be returned to the requester.

Before we can test we need to create a new class in a new package. Create a package called com.ibm.itso.json.model. Within this package create a new class called Product, as shown in Example 7-18.

*Example 7-18 Product class*

---

```
public class Product {
    Stringid;
    Stringname;
    int stock;
    double price;
    floatweight;
}
```

---

To complete the object creation Generate the Getters and Setters for this class using **Context Menu** → **Source** → **Generate Getters and Setters**.

We now add code to the servlet that will take the product class POJO and serialize it using the method just added. Add the code shown in Example 7-19 to the servlet in the area where the other code snippets have been added. You can see that the code contains different types and test data.

*Example 7-19 Code snippet to turn a POJO into JSON*

---

```
else if (path.equals("/pojo")) {  
  
    Product prod = new Product();  
  
    prod.setId("12345");  
    prod.setName("");  
    prod.setPrice(149.99d);  
    prod.setStock(200);  
    prod.setWeight(11.2f);  
  
    writeJSONObject(res, new Object[]{prod});  
  
}
```

---

Once these elements have been added save the servlet and then rerun the servlet test harness index.html. Select the item labelled POJO. You should see the result in the Web browser shown in Figure 7-15.



*Figure 7-15 Result from POJO object serialization*

You can see that the Plain Old Java Object has been introspected into a JSONObject instance and this in turn has been serialized into the JSON data stream.

### 7.3.12 JSON4J limitations

The JavaScript Object Notation string parsing adheres to the guidelines presented by multiple reference sources on the format of JSON text. Primarily, according to all sources reviewed, attribute names are always quoted strings. See the JSON structure shown in Example 7-20.

*Example 7-20 Example JSON structure*

---

```
{  
  "attributeName": "foo"  
}
```

---

However, for browsers that support attribute names without quotation marks, attribute names still evaluate correctly. The JSON4J library follows the definition of the JSON format and is more rigid than most browsers. The library will not evaluate unquoted attribute names as valid json. Instead, JSON4J will generate a `java.io.IOException` exception. Therefore, always use correctly quoted attribute names in JSON text that you pass into the parse functions of this library.

## 7.4 Feeds libraries (Apache Abdera)

IBM WebSphere Web 2.0 Feature Pack has packaged the Apache Abdera Java libraries in the Abdera-based FeedSphere libraries support. Apache Abdera is an open source project providing feeds support. The Apache Abdera library provides an implementation of the IETF Atom Syndication Format and Atom Publishing Protocol standards (RFCs 4287 and 5023). The Apache Abdera library provides support for the reading of RSS content. You can view the detail specification at:

<http://www.ietf.org/rfc/rfc4287>

Support for Web feeds is included as in the Web 2.0 to SOA connectivity. Atom and RSS libraries allow you to expose JEE resources as Web 2.0 style “data feeds”. This permits clients to subscribe to updates and be notified when the data changes, provided the update frequency does not approach near-real-time. For real-time notifications, the Web messaging service (see Chapter 10, “Web messaging service” on page 455) would be a more appropriate package to use.

Web feeds allow software programs to check for updates published on a Web site. Feeds can be dynamically generated by a Web site and are often used with content management systems or news syndication systems.

To provide a Web feed, a Web site uses specialized software that publishes a list (or *feed*) of recent articles or content in a standardized, machine-readable format. The feed can then be downloaded by Web sites that syndicate content from the feed, or by feed reader programs that allow Internet users to subscribe to feeds and view their content. A feed contains entries, which may be headlines, full-text articles, excerpts, summaries, and links to content on a Web site, along with various metadata.

### 7.4.1 Package structure

The feeds libraries are distributed in three ways:

- ▶ As a stand-alone package
- ▶ As part of the IBM WebSphere Application Server Feature Pack for Web 2.0 installer
- ▶ As an Eclipse plug-in

#### Stand-alone package

The stand-alone Abdera jar files are packaged within the lib directory, as shown in Figure 7-16.

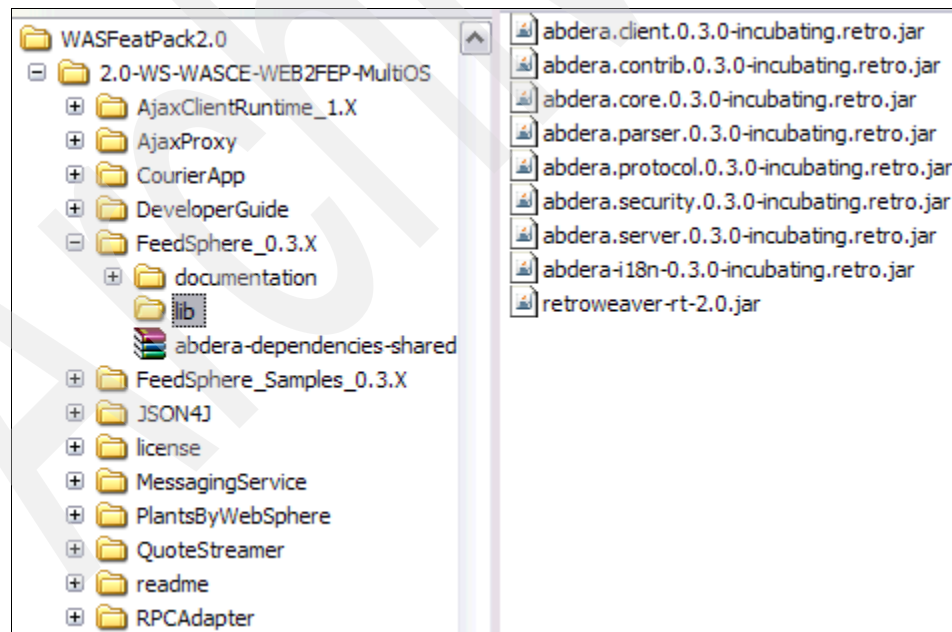


Figure 7-16 The stand-alone Abdera JARs

The Abdera dependency jars are packaged as shown in Figure 7-17.

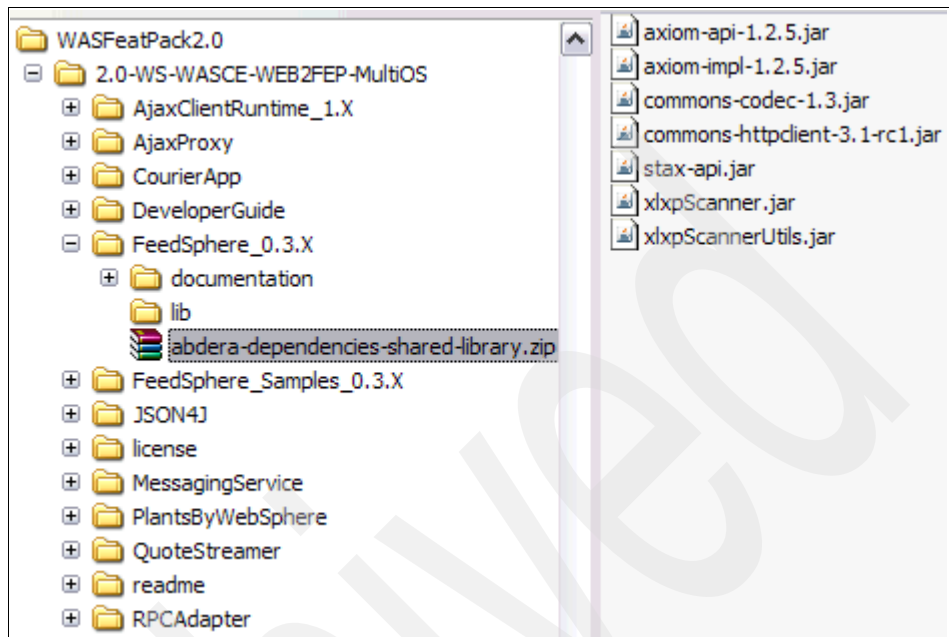


Figure 7-17 The stand-alone Abdera dependency JARs

## IBM WAS Feature Pack for Web 2.0 installation

The FeedSphere library contains all of the Abdera jars, as shown in Figure 7-18.

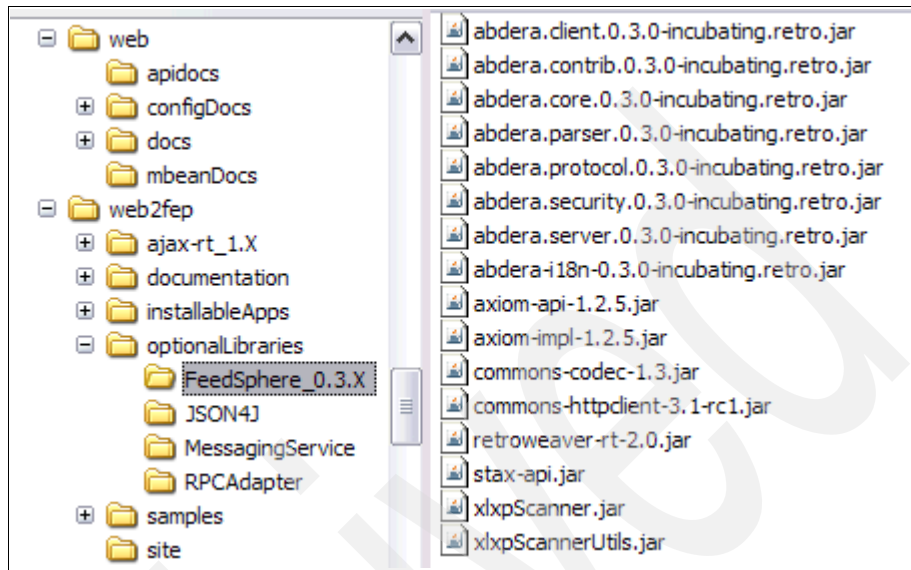


Figure 7-18 Feeds libraries as part of the Feature Pack for Web 2.0 WAS installation

### 7.4.2 Atom 1.0 Syndication Format

The Atom Syndication Format (RFC 4287) can be found at:

<http://www.ietf.org/rfc/rfc4287.txt>

Web content syndication has evolved into a platform for next-generation Web-based services and content distribution. While the adoption of syndication technologies has recently progressed rapidly, these technologies have a long history of technical issues, ambiguities, and interoperability challenges that have made life difficult for software developers and consumers attempting to use these emerging trends. To address these issues, members of the syndication technology community came together to pool their combined experiences and define the Atom Syndication Format and the Atom Publishing Protocol (APP) standards. Atom requires that every feed and entry contain three elements:

- ▶ A unique identifier, which can be as simple as the URI of a blog entry, or as complex as a 128-bit Globally Unique Identifier (GUID).
- ▶ A title that expresses a human readable subject line for the entry, or it can be a blank string (represented by an empty title element, that is, <title />).
- ▶ A time stamp that indicates when the last update occurred.

Example 7-21 shows an example of an Atom feed.

*Example 7-21 Atom feed example*

---

```
<feed xmlns="http://www.w3.org/2005/Atom"
      xml:lang="en"
      xml:base="http://www.example.org">
  <id>http://www.example.org/myfeed</id>
  <title>A Sample Feed</title>
  <updated>2008-06-16T18:34:42Z</updated>
  <link href="/blog" />
  <link rel="self" href="/myfeed" />
  <entry>
    <id>http://www.example.org/entries/1</id>
    <title>A sample blog entry</title>
    <link href="/blog/2008/06/1" />
    <updated>2008-06-16T18:34:42Z</updated>
    <summary>This is a sample blog entry</summary>
  </entry>
  <entry>
    <id>http://www.example.org/entries/2</id>
    <title />
    <link href="/blog/2008/06/2" />
    <updated>2008-06-16T18:34:42Z</updated>
    <summary>This is sample blog entry without a title</summary>
  </entry>
</feed>
```

---

Atom also describes a flexible, consistent content model that is capable of supporting:

- ▶ Plain text
- ▶ Escaped HTML
- ▶ Well-formed XHTML
- ▶ Arbitrary XML
- ▶ Base-64 encoded binary content
- ▶ URI pointers to content not included directly within the feed

In contrast, without resorting to the use of non-standardized and therefore inconsistently implemented namespace extensions, RSS is capable of only handling plain text and escaped HTML content.

Atom also provides a well-defined extensibility model. This model provides decentralized, dynamic mechanisms for adding new metadata and content supported by RSS, but does so in a way that helps protect core interoperability between implementations. For instance, Atom clearly outlines where extension

elements can and cannot appear within a document, which extensions are language sensitive (and thereby affected by `xml:lang` attributes), and how an Atom implementation must react when it encounters an unfamiliar extension element. Finally, Atom provides rigorous definitions for the various required and optional metadata elements within its core namespace, such as an author element that is a complex structure including a name, an e-mail address, and a resource identifier that is associated with the author in some way (for example, URI).

Atom 1.0 also includes:

- ▶ The ability for individual entries to exist independently of a feed
- ▶ ISO-8601 and XML Schema compatible timestamps
- ▶ Relative URI support using `xml:base`
- ▶ Enhanced internationalization through the use of Internationalized Resource Identifiers (IRIs) and `xml:lang`
- ▶ Accessibility features that make it easier for users with disabilities to consume feeds
- ▶ An HTML-like dynamically extensible link mechanism that links a feed or entry to external resources
- ▶ Self-referential feeds
- ▶ A MIME media type that can identify Atom 1.0 documents
- ▶ Built-in support for XML Digital Signatures and XML Encryption
- ▶ A non-normative RELAXNG schema that validates Atom 1.0 document instances
- ▶ A core subset that is compatible with RDF

Currently, complex media, such as podcasting, is enabled through the use of RSS 2.0's enclosure tag. However, there are several problems with this solution:

- ▶ RSS does not allow multiple enclosures within an entry (meaning multiple formats, such as MP3 or WMA, must offer separate feeds for each format offered).
- ▶ RSS does not provide a means of associating a language with the enclosed resource.
- ▶ RSS enclosures are required to use HTTP URLs.
- ▶ RSS does not provide a means of optionally associating a human-readable title for a referenced resource.



Atom allows any single entry to contain multiple enclosures, each with an associated media type attribute that makes it possible to produce a single feed containing all of the formats to distribute.

Example 7-22 shows the multiple enclosure links.

*Example 7-22 Atom podcasting example with enclosure links*

---

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <id>http://www.example.org/myfeed</id>
  <title>My Podcast Feed</title>
  <updated>2008-06-16T18:34:42Z</updated>
  <author>
    <name>Robin Howlett</name>
  </author>
  <link href="http://example.org" />
  <link rel="self" href="http://example.org/myfeed" />
  <entry>
    <id>http://www.example.org/entries/1</id>
    <title>Atom 1.0</title>
    <updated>2008-06-16T18:34:42Z</updated>
    <link href="http://www.example.org/entries/1" />
    <summary>An overview of Atom 1.0</summary>
    <link rel="enclosure"
      type="audio/mpeg"
      title="MP3"
      href="http://www.example.org/myaudiofile.mp3"
      length="1234" />
    <link rel="enclosure"
      type="application/x-bittorrent"
      title="BitTorrent"
      href="http://www.example.org/myaudiofile.torrent"
      length="1234" />
    <content type="xhtml">
      <div xmlns="http://www.w3.org/1999/xhtml">
        <h1>Show Notes</h1>
        <ul>
          <li>00:01:00 -- Introduction</li>
          <li>00:15:00 -- Talking about Atom 1.0</li>
          <li>00:30:00 -- Summary</li>
        </ul>
      </div>
    </content>
  </entry>
</feed>
```

---

Atom link elements that enable enclosures can do much more than associate downloadable files with an entry. Links can also specify meaningful links to other types of resources:

- ▶ `<link rel="alternate" />`

This identifies an alternate version of the feed or entry (for example, a Weblog home page).

- ▶ `<link rel="related" />`

This identifies a resource that is described in some way by the content of the entry.

- ▶ `<link rel="self" />`

This identifies a resource that is equivalent to the feed or entry. Generally, this permits a feed or entry to become self-referential to allow flexible auto-discovery mechanisms.

- ▶ `<link rel="via" />`

This identifies a resource that provided the information contained in the feed or entry. For example, if the entry was distributed through an online aggregation service, the *via* link identifies the aggregator as an alternative to a common practice of having the aggregator override the RSS link element.

These built-in link relations are designed to cover the most common types of links used with feeds.

Atom has the ability to reference entry content by URI. This content-by-reference mechanism provides a flexible means for expanding the types of content that can be syndicated using Atom, such as photo blogs, streaming video links, or software updates.

### 7.4.3 Atom Publishing Protocol (APP)

The Atom Publishing Protocol (RFC 5023) can be seen at:

<http://www.ietf.org/rfc/rfc5023.txt>

The Atom Publishing Protocol is an HTTP-based approach for creating and editing Web resources. It is designed to use the basic operations provided by the HTTP protocol (such as GET, PUT, and DELETE) to pass instances of Atom 1.0 feed and entry documents that represent such things as blog entries, podcasts, wiki pages, calendar entries, and so on.

Central to the Atom Publishing Protocol is the concept of collections of editable resources that are represented by Atom 1.0 feed and entry documents. A collection has a unique URI. Issuing an HTTP GET request to that URI returns

an Atom feed document. To create new entries in that feed, clients send HTTP POST requests to the collection's URI. Those newly created entries will be assigned their own unique edit URI.

To modify an entry, the client simply retrieves the resource from the collection, makes the modification, and puts it back. Removing an entry from the feed is a simple matter of issuing an HTTP DELETE request to the appropriate edit URI. All operations are performed using simple HTTP requests, which are usually performed with nothing more than a simple text editor and a command prompt.

An explanation of how APP can be used with Apache Abdera is provided using the FeedSphere sample application.

## 7.4.4 Apache Abdera

Abdera was developed by the Apache organisation and has been designed as a Java library that can be utilized by Java applications on both the client and the server. The Abdera library that is a component of the Feature Pack for Web 2.0 is composed of a number of individual modules:

- Core (abdera.core.0.3.0-incubating.retro.jar)

This provides the core interfaces used throughout Abdera, which include:

- Feed Object Model

A set of Java interfaces and abstract classes modeled after the various document and element types defined by the Atom specifications.

- Parser and factory interfaces

Interfaces that provide the mechanisms for creating and consuming Atom documents.

- ExtensionFactory

The primary means by which extensions to the Atom format are integrated into Abdera. ExtensionFactories are used by both the parser and the factory.

- Writer

The means by which feed object model objects are serialized into XML or other formats.

- ParseFilter

Provides a means of filtering the stream of parse events.

- Converter

Base interfaces and utilities for converting objects into feed object model objects.

– XPath

Base interface for navigating the feed object model using XPath.

- ▶ Parser (abdera.parser.0.3.0-incubating.retro.jar)  
Provides an implementation of the feed object model based on the Apache Axiom project.
- ▶ Protocol (abdera.protocol.0.3.0-incubating.retro.jar)  
Provides the base interfaces and utility classes used for the RFC 5023 (AtomPub) implementation.
- ▶ Server (abdera.server.0.3.0-incubating.retro.jar)  
Provides framework code used to build AtomPub servers.
- ▶ Client (abdera.client.0.3.0-incubating.retro.jar)  
Provides an AtomPub client implementation that is based on the Apache Commons HTTP Client.
- ▶ Security (abdera.security.0.3.0-incubating.retro.jar)  
Provides support for XML Digital Signatures and XML Encryption.
- ▶ i18n (abdera.i18n.0.3.0-incubating.retro.jar)  
Provides support for internationalization.
- ▶ Contrib (abdera.contrib.0.3.0-incubating.retro.jar)  
Provides support for reading RSS feeds.

If Java version 1.4 is being used, the Retroweaver package provides retrospective mapping (retroweaver-rt-2.0.jar (required for JDK™ 1.4 support)).

**Note:** Most of the core Abdera components (that is, Abdera, factory, and parser objects) are thread-safe and stateless. The feed object model (FOM) objects that they create are, however, not thread-safe. Therefore, if necessary, you must perform your own synchronization.

## Dependencies

Abdera requires certain resources for the core and protocol. The dependencies supplied with the feature pack are:

- ▶ Axiom (required by core)
  - axiom-api-1.2.5.jar
  - axiom-impl-1.2.5.jar
- ▶ Apache Commons
  - Codec: commons-codec-1.3.jar (required by protocol)
  - HttpClient: commons-httpclient-3.1-rc1.jar (required by client)
- ▶ StAX (any StAX API impl is required at run time)  
stax-api.jar
- ▶ SAX Parsing
  - xlpScanner.jar
  - xlpScannerUtils.jar
- ▶ Jaxen (support for XPath)  
jaxen-1.1.1.jar (required by the core) (available at <http://jaxen.codehaus.org/releases.html>)

### 7.4.5 FeedSphere sample

The FeedSphere sample has been built to demonstrate the Atom and RSS support in Abdera. In addition, it also demonstrates the Atom Publishing Protocol support. The specific scenarios that they support are

- ▶ Generate an Atom feed.  
A simple example to create an Atom feed.
- ▶ Read an Atom feed.  
An example to read Atom content. Also, perform a filtered read on the Atom content. For example, select an Atom feed from the O'Reilly Web site:  
<http://www.oreillynet.com/pub/feed/1>

- ▶ Read an RSS feed.

An example for reading RSS content. For example, choose an RSS feed from:

[http://rss.cnn.com/rss/cnn\\_topstories.rss](http://rss.cnn.com/rss/cnn_topstories.rss)

- ▶ Atom Publishing Protocol support.

Demonstrates the ability to create, update, delete, and retrieve entries in a feed. Also provides options to retrieve the service document for the deployed Abdera server and the associated feed.

### **Limitations of FeedSphere samples**

The FeedSphere sample application demonstrates a subset of the features available in the Abdera libraries. Therefore, the limitations of FeedSphere are those of Abdera. For example, Abdera supports reading or parsing of RSS feeds, but does not have the ability to generate RSS feeds.

## FeedSphere sample installation through WebSphere

The FeedSphereApp.war file location is dependent on the package that you download:

- ▶ If the WAS Feature Pack for Web 2.0 was installed as an Eclipse feature, then you can locate the WAR in the `<ECLIPSE_HOME>/plugins/com.ibm.ajax.feedsphere.samples_1.0.0/FeedSphereApp.war` directory.
- ▶ If you downloaded the stand-alone compressed file, the FeedSphereApp.war is in the extracted location (see Figure 7-19).

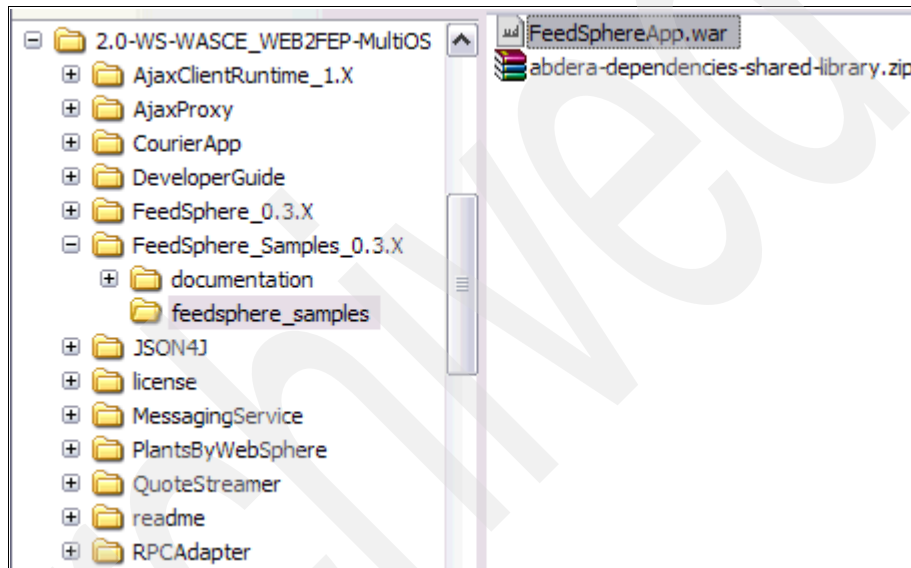


Figure 7-19 The FeedSphere sample WAR file

In the case of WebSphere Application Server 6.x, the previously mentioned JAR files must be added into the class path. As a recap, the JAR files referred earlier are of the pattern axiom-\*.jar, xlxp\*.jar, stax-api.jar, commons-\*.jar, jaxen-\*.jar.

One mechanism is to create a shared library, and then associate it with the applications that require those JAR files. The following steps illustrate the creation of a shared library, and then associating the library with the application:

1. Create a folder for the shared library option and name it the feedsphere-library.
2. Copy the following jar files into the feedsphere-library folder:
  - axiom-api-1.2.5.jar
  - commons-httpclient-3.1-rc1.jar
  - xlpScannerUtils.jar
  - axiom-impl-1.2.5.jar
  - stax-api.jar
  - commons-codec-1.3.jar
  - xlpScanner.jar
  - jaxen-1.1.1.jar
3. Open the WebSphere Application Server administration console and select the **Feed Sample** application. Select **Applications** → **Enterprise Applications** → **FeedSphereApp\_war**.
4. Click the **Shared library** link available there. Select the check box for **FeedSphereApp\_war**, and select the **Reference shared libraries** button.
5. Register the shared library, if it is not already present. Click **New**, and on the Shared Library Mapping → New page, enter **fslib** for the shared library name, and the classpath to the shared libraries. As an example, in this case:
  - /feedsphere-library/axiom-api-1.2.5.jar
  - /feedsphere-library/axiom-impl-1.2.5.jar
  - /feedsphere-library/stax-api.jar
  - /feedsphere-library/xlpScanner.jar
  - /feedsphere-library/xlpScannerUtils.jar
  - /feedsphere-library/commons-httpclient-3.1-rc1.jar
  - /feedsphere-library/commons-codec-1.3.jar
  - /feedsphere-library/jaxen-1.1.1.jar
6. Save this information by clicking **OK**. The page switches to the Shared library mapping page. Select the **fslib** library and move it to the Selected list box. Click **OK**.
7. This returns you to the Shared library references page. Select the check box for **FeedSphereApp\_war** and select **OK**.
8. Save all these changes to the master configuration.
9. Start the feed sample application.



## FeedSphere sample installation using Rational Application Developer

To install the FeedSphere sample using Rational Application Developer:

1. Open Rational Application Developer in a new workspace and then close the Welcome page.
2. Click **File** → **New** → **Other** from the workbench toolbar.
3. In the New Project window, expand **Examples** → **IBM WebSphere Application Server Feature Pack for Web 2.0 samples**.
4. Select **FeedSphere Application Sample** and click **Next**, as shown in Figure 7-20.

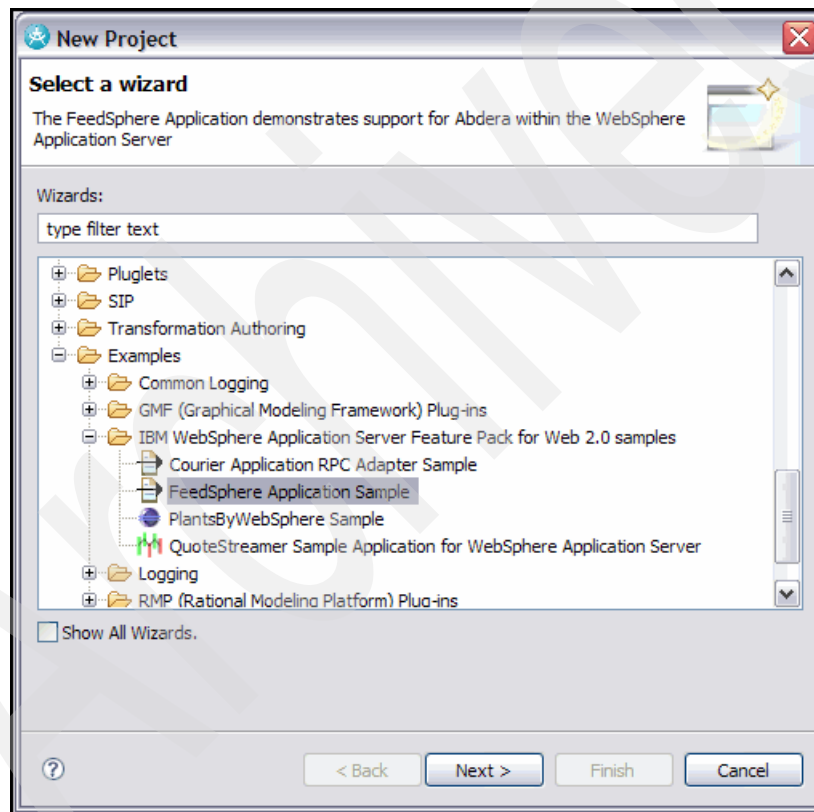


Figure 7-20 New Project wizard

5. Select **WebSphere Application Server v6.1** as the target run time from the drop-down list. We deploy this sample on a WebSphere test environment installed with the installation of Rational Application Developer. Click **Finish**, as shown in Figure 7-21.

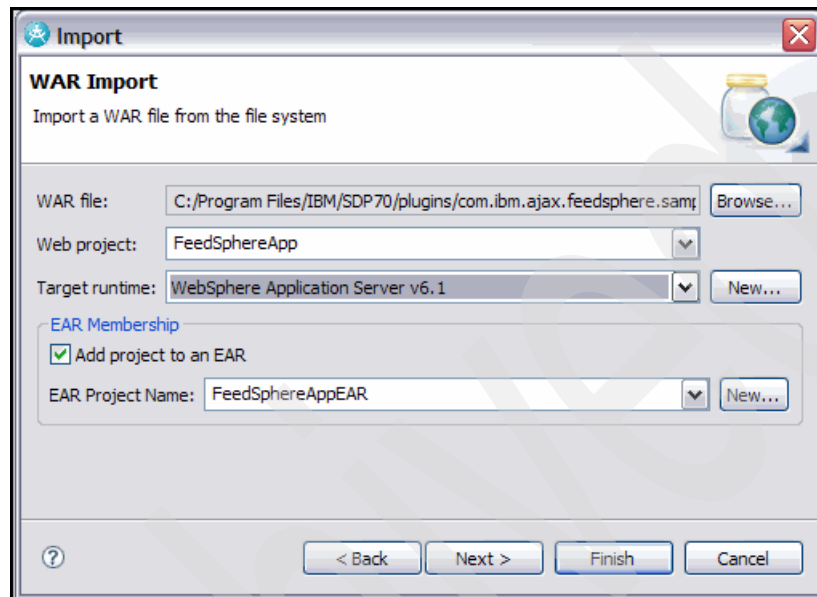


Figure 7-21 Import WAR wizard

6. The complete EAR project with the supported Web module will be imported and we can browse through modules (FeedSphereAppEar - EAR module and FeedSphereApp - Web module) in the Project Explorer view, as shown in Figure 7-22. (You may notice some errors. These are because of some missing supporting library files. We take care of these errors in next few steps.)

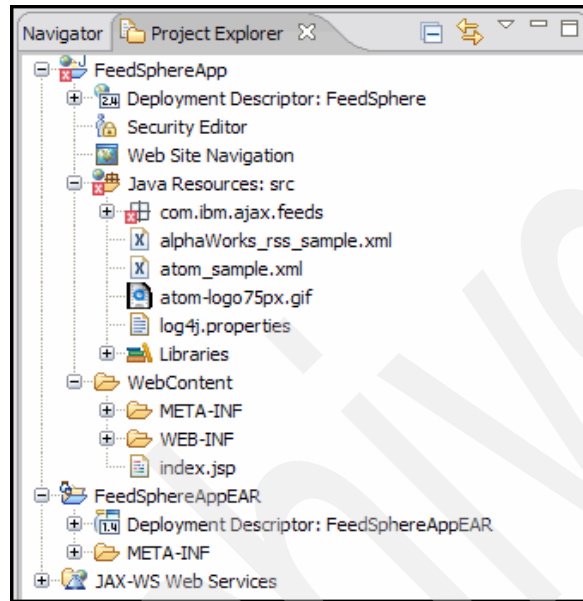


Figure 7-22 FeedSphere Application sample project structure

7. Browse to <<Installation Directory>> \Web2fep\samples\FeedSphere\_0.3.X and extract the abdera-dependencies-shared-library.zip file contained within this folder. You should see seven jar files within the folder. They are:

- Axiom JAR files:
  - axiom-api-1.2.5.jar
  - axiom-impl-1.2.5.jar
- Stax API support:
  - xlxpScannerUtils.jar
  - xlxpScanner.jar
  - stax-api.jar

These files are for WebSphere Application Server 6.0.x and 6.1.x.

- Support for the Abdera client calls:
  - commons-httpclient-3.1-rc1.jar
  - commons-codec-1.3.jar

- Support for XPath Implementation: jaxen-1.1.1.jar.  
(If you are not getting this file in the folder, you can get it from:  
<http://repo1.maven.org/maven2/jaxen/jaxen/1.1.1/>)
- 8. Expand **FeedSphereApp** → **WebContent** → **WEB-INF** in Project Explorer. Right-click the **lib** folder in Project Explorer and select **Import**.
- 9. The Import Pop-up window now appears. Select **General** → **File System** → **Next**, as shown in Figure 7-23.

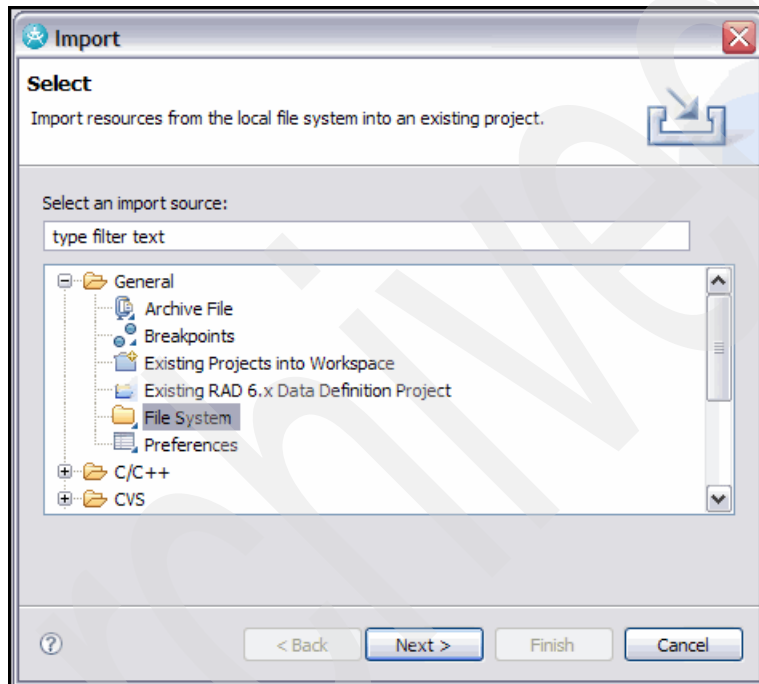


Figure 7-23 Import from file system

10. Browse for the folder into which you extracted all of the jar files (in our case <<Installation Directory>>\Web2fep\samples\FeedSphere\_0.3.X), select all of the jar files, and click **Finish**, as shown in Figure 7-24.

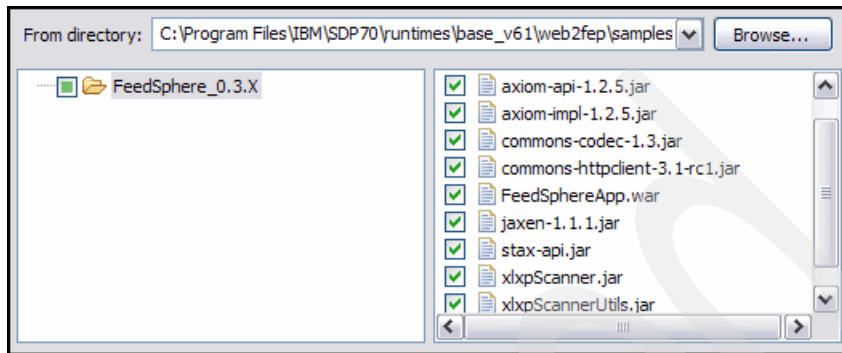


Figure 7-24 Select supporting jar files

11. All supported jar files will now be imported into the lib folder and any errors previously reported in Project Explorer should no longer appear.
12. Start WebSphere Application Server v6.1 from the Servers view. The server status will change to started.
13. Right-click **WebSphere Application Server v6.1** in the Servers view and select **Add and Remove Projects**.
14. Select **FeedSphereAppEAR** in Available Projects and click **Add** to move it in to the Configured Projects. Click **Finish**.
15. Expand **FeedSphereApp** → **WebContent** → **WEB-INF** in Project Explorer. Right-click **index.jsp** and select **Run As** → **Run On Server**.
16. Click **Finish** on the pop-up window.
17. You should be able to see the contents of <http://localhost:9080/FeedSphereApp/index.jsp> in the browser window.

## 7.4.6 Atom Publishing Protocol and HTTP operations

For Hypertext Transfer Protocol - HTTP/1.1 (RFC 2616) see:

<http://www.w3.org/Protocols/rfc2616/rfc2616.html>

The Atom Publishing Protocol's central concept is collections of editable resources that are represented by Atom 1.0 feed and entry documents. A collection has a unique URI. Issuing an HTTP GET request to that URI returns an Atom feed document. To create new entries in that feed, clients send HTTP POST requests to the collection's URI. Those newly created entries will be

assigned their own unique edit URI. To modify those entries, the client simply retrieves the resource from the collection, makes its modifications, then puts it back. Removing the entry from the feed is a simple matter of issuing an HTTP DELETE request to the appropriate edit URI. All operations are performed using simple HTTP requests and can usually be performed with nothing more than a simple text editor and a command prompt.

There are two approaches for performing these operations:

- ▶ Use the XMLHttpRequest (XHR) object in the browser to perform the PUT, POST, GET, and DELETE operations. However, the normal form submit operation in the browser supports only the GET and POST operations.
- ▶ Use a tool like cURL. cURL provides a simple command-line operation to invoke calls on the APP server.

```
curl -s -X POST --data-binary @entry.xml  
http://example.org/atom/entries
```

```
curl -s -X GET http://example.org/atom/entries/1
```

```
curl -s -X PUT --data-binary @entry.xml  
http://example.org/atom/entries/1  
curl -s -X DELETE http://example.org/atom/entries/1
```

For more information see:

<http://curl.haxx.se/download.html>

## Creating an Atom feed

After selecting the hyperlink `/FeedSphereApp/feed?create` to create the sample atom feed the request is sent to the FeedSphere servlet. See Figure 7-25.



Figure 7-25 Web front end for creating an Atom feed

The Web.xml file deployed with the FeedSphere sample application maps any URL containing *feed* to the FeedSphere class. See Figure 7-26.

```
<!-- FeedSphere Servlet for generating-consuming feeds -->
<servlet id="feed">
    <description></description>
    <display-name>FeedSphere</display-name>
    <servlet-name>FeedSphere</servlet-name>
    <servlet-class>
        com.ibm.ajax.feeds.samples.FeedSphere
    </servlet-class>
</servlet>
<servlet-mapping id="FeedSphere">
    <servlet-name>FeedSphere</servlet-name>
    <url-pattern>/feed/*</url-pattern>
</servlet-mapping>
```

Figure 7-26 Web.xml shows a URL containing *feed* mapped to *FeedSphere* class

The query parameters passed on the request are parsed and the *?create* paramater is mapped to the *generateDefaultFeed* method in the *CreateAtomFeedContent* class. See Example 7-23.

*Example 7-23 generateDefaultFeed()*

---

```
public Feed generateDefaultFeed() {
    Factory factory = Abdera.getNewFactory();
    Feed feed = factory.newFeed();
    feed.setTitle("IBM alphaWorks Emerging Technologies : ");
    feed.addLink("http://localhost:8080/");
    feed.setUpdated(new Date());
    feed.addAuthor("alphaWorks Author");
    feed.setId("http://www.alphaworks.ibm.com/");
    feed.addCategory("technology-updates");

    Entry entry1 = feed.insertEntry();
    entry1.setTitle("Web 2.0 Starter Toolkit for DB2");
    entry1.addLink("/2007/Web20starterkit");
    entry1.setId("Web2db2");
    entry1.setUpdated(new Date());
    entry1.setSummary("A conveniently packaged set of products and
technologies that enable the quick creation of DB2 Web services and
feeds using PHP technology. (NEW: 07/26/2007 in home)");

    Entry entry2 = feed.insertEntry();
```

```
entry2.setTitle("Real-Time Active Inference and Learning (RAIL)");
entry2.addLink("/2007/rail");
entry2.setId("rail");
entry2.setUpdated(new Date());
entry2.setSummary("A Bayesian inference tool using active probing
for real-time, adaptive problem diagnosis in distributed systems. (NEW:
07/17/2007 in home)");
return feed;
}
```

---

This example illustrates all of the key characteristics of the feed object model. First, you acquire an instance of `org.apache.abdera.factory.Factory`. This is the interface that bootstraps the creation of all feed object model objects. You create a new instance of `org.apache.abdera.model.Feed` using the factory and set the feed object's attributes. The **`insertEntry()`** command on the feed object creates an `Entry` that is then populated. The returned feed object is return so that the document is written and the feed is created.



Using cURL we can view the created Atom feed document shown in Figure 7-27.

```
c:\dev\ibm>curl -v -X GET http://localhost:9080/FeedSphereApp/feed?create
* About to connect() to localhost port 9080 (#0)
* Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 9080 (#0)
> GET /FeedSphereApp/feed?create HTTP/1.1
> User-Agent: curl/7.18.1 (i386-pc-win32) libcurl/7.18.1 OpenSSL/0.9.8g zlib/1.2.3
> Host: localhost:9080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Language: en-IE
< Transfer-Encoding: chunked
< Date: Thu, 26 Jun 2008 21:52:47 GMT
< Server: WebSphere Application Server/6.1
<
<?xml version="1.0" encoding="UTF-8"?><feed xmlns="http://www.w3.org/2005/Atom"><title type="text">IBM alphaWorks Emerging Technologies : </title><link href="http://localhost:9080/"></link><updated>2008-06-26T21:52:48.578Z</updated><author><name>alphaWorks Author</name></author><id>http://www.alphaworks.ibm.com/</id><category term="technology-updates"></category><entry><title type="text">Real-Time Active Inference and Learning (RAIL)</title><link href="/2007/rail"></link><id>rail</id><updated>2008-06-26T21:52:48.578Z</updated><summary type="text">A Bayesian inference tool using active probing for real-time, adaptive problem diagnosis in distributed systems. (NEW: 07/17/2007 in home)</summary></entry><entry><title type="text">Web 2.0 Starter Toolkit for DB2</title><link href="/2007/web20starterkit"></link><id>web20h2</id><updated>2008-06-26T21:52:48.578Z</updated><summary type="text">A conveniently packaged set of products and technologies that enable the quick creation of DB2 Web services and feeds using PHP technology. (NEW: 07/26/2007 in home)</summary></entry></feed>* Connection #0 to host localhost left intact
* Closing connection #0
```

Figure 7-27 Using cURL

Note that the XML namespace declarations and date formats are handled automatically by the feed object model implementation. However, the serialized output provided by the cURL tool may be difficult to read due to the lack of line breaks and indentation.

To obtain the output shown in Figure 7-28 on page 198 you can use the prettyxml writer to insert line breaks and indents, as follows:

```
Writer writer = abdera.getWriterFactory().getWriter("prettyxml");
writer.writeTo(entry, System.out);
```

```

- <feed>
  <title type="text">IBM alphaWorks Emerging Technologies : </title>
  <link href="http://localhost:8080/">
  <updated>2008-06-26T22:39:34.078Z</updated>
  - <author>
    <name>alphaWorks Author</name>
  </author>
  <id>http://www.alphaworks.ibm.com/</id>
  <category term="technology-updates"/>
  - <entry>
    <title type="text">Real-Time Active Inference and Learning
    (RAIL)</title>
    <link href="/2007/rail/">
    <id>rail</id>
    <updated>2008-06-26T22:39:34.078Z</updated>
    - <summary type="text">
      A Bayesian inference tool using active probing for real-time, adaptive
      problem diagnosis in distributed systems. (NEW: 07/17/2007 in
      home)
    </summary>
  </entry>
  - <entry>
    <title type="text">Web 2.0 Starter Toolkit for DB2</title>
    <link href="/2007/web20starterkit/">
    <id>web2db2</id>
    <updated>2008-06-26T22:39:34.078Z</updated>
    - <summary type="text">
      A conveniently packaged set of products and technologies that
      enable the quick creation of DB2 Web services and feeds using PHP
      technology. (NEW: 07/26/2007 in home)
    </summary>
  </entry>
</feed>

```

Figure 7-28 The formatted output

## Reading an Atom feed

The feed just created can now be read programmatically. Let us use the following URL:

<http://localhost:9080/FeedSphereApp/feed?read=http://localhost:9080/FeedSphereApp/feed?create>

The servlet passes the URL to the `readContent` method shown in Figure 7-29.

```
private String readContent(String url) {  
    String returnValue = NO_CONTENT;  
    ReadFeedContent workingInstance =  
        new ReadFeedContent();  
    InputStream atomInputStream =  
        workingInstance.generateAtomStream(url);  
    if (atomInputStream != null)  
        returnValue = workingInstance.  
            parseContent(atomInputStream);  
    return returnValue;  
}
```

Figure 7-29 *readContent* method

The parseContent method shown in Figure 7-30 demonstrates parsing of the feed.

```
public String parseContent(  
    InputStream atomInputStream) {  
    StringBuffer parsedContent = new StringBuffer();  
    Document doc =  
        Abdera.getNewParser().parse(atomInputStream);  
    Feed feed = (Feed) doc.getRoot();  
    List list = feed.getEntries();  
    parsedContent.  
        append("--Summary-----\n");  
    parsedContent.append("Feed created by: ");  
    if (null != feed.getAuthor())  
        parsedContent.  
            append(feed.getAuthor().getName());  
    else  
        parsedContent.append("Author not listed");  
    parsedContent.append(END_OF_LINE);  
    for (int i = 0; i < list.size(); i++) {  
        Entry listElement = (Entry) list.get(i);  
        parsedContent.append("# ").  
            append(listElement.getTitle());  
        parsedContent.append(" :: ");  
        parsedContent.append(listElement.getSummary());  
        parsedContent.append(" :: ");  
        parsedContent.append(listElement.getUpdated());  
        parsedContent.append(END_OF_LINE);  
    }  
  
    return parsedContent.toString();  
}
```

Figure 7-30 parseContent method

The InputStream object is parsed by Abdera into a document object that in turn returns a feed object representing the feed. The feed's entries are next stored in a list by calling the feed.getEntries() method, and other data such as the author is accessible through the feed object. The list of entry objects is iterated to extract the title, summary, and last updated time data, which are returned in the StringBuffer object.

The returned output is shown Figure 7-31.

```
--Summary-----  
Feed created by: alphaWorks Author  
# Real-Time Active Inference and Learning (RAIL) :: A Bayesian  
inference tool using active probing for real-time, adaptive  
problem diagnosis in distributed systems. (NEW: 07/17/2007 in  
home) :: Fri Jun 27 02:12:05 BST 2008  
# Web 2.0 Starter Toolkit for DB2 :: A conveniently packaged set  
of products and technologies that enable the quick creation of  
DB2 Web services and feeds using PHP technology. (NEW:  
07/26/2007 in home) :: Fri Jun 27 02:12:05 BST 2008
```

Figure 7-31 The consumed feed

**Tip:** As an alternative, XPath statements can be used to navigate parsed documents. Also, Abdera's feed object model can be used with XSLT to apply transforms to parsed documents and elements.

## Filtering an Atom feed

Similarly, you can filter the content of a feed while parsing (Figure 7-32):

<http://localhost:9080/FeedSphereApp/feed?filter=http://localhost:9080/FeedSphereApp/feed?create>

```
public String parseContent(InputStream atomInputStream) {
    StringBuffer parsedContent = new StringBuffer();
    Parser abderaParser = Abdera.getNewParser();
    // Set the whitelist filter options
    ParserOptions whitelistOption =
        abderaParser.getDefaultParserOptions();
    ListParseFilter filter =
        new WhiteListParseFilter();
    filter.add(Constants.FEED);
    filter.add(Constants.ENTRY);
    filter.add(Constants.TITLE);
    whitelistOption.setParseFilter(filter);
    abderaParser.
        setDefaultParserOptions(whitelistOption);

    Document doc =
        abderaParser.parse(atomInputStream);
    Feed feed = (Feed) doc.getRoot();
    List list = feed.getEntries();
    parsedContent
        .append("--Summary-----\n");
    for (Iterator iterator = list.iterator();
        iterator.hasNext();) {
        Entry listElement = (Entry) iterator.next();
        parsedContent.append("# ").
            append(listElement.getTitle());
        parsedContent.append(END_OF_LINE);
    }

    return parsedContent.toString();
}
```

Figure 7-32 Filtering an Atom feed

By supplying the Abdera Parser with the specified ParserOptions, it is easy to limit the parsing to only the desired feed elements, as shown in Figure 7-33.

```
--Summary-----  
# Real-Time Active Inference and Learning (RAIL)  
# Web 2.0 Starter Toolkit for DB2
```

Figure 7-33 Filtered output

Abdera's feed object model interfaces follow the Atom Syndication Format schema very closely, making it easy for developers who are familiar with the Atom Syndication Format specification (RFC 4287) to produce valid Atom documents. However, it is important to point out that the implementation does not perform any validation of the input. For instance, RFC 4287 requires that all atom:entry and atom:feed elements contain exactly one atom:id element whose value is a normalized IRI.

See Internationalized Resource Identifiers (RFC 3987) at:

<http://www.ietf.org/rfc/rfc3987.txt>

Abdera, however, does not throw an exception if a developer tries to create and serialize an entry or feed that does not contain an atom:id or that contains multiple atom:ids. Developers are responsible for ensuring that the documents that they produce are valid.

### **Adding extensions and content flexible**

The FOM also has the ability to work with extensions to the Atom format (that is, elements from the OpenSearch specification). Atom entries can include plain-text, XML, or any character-based data. It can also reference content by URIs, all of which can be set with the Entry's setContent method.

### **Working with RSS feeds**

This same code can also read RSS feeds. The caveat, however, is that the supplied Abdera libraries cannot create or filter an RSS feed.

You can check the validity of Atom documents with the FeedValidator at:

<http://www.feedvalidator.org>

## **7.4.7 AtomPub server implementation using Abdera**

Before we introduce interacting with an Abdera-based APP service, we will introduce the central concepts behind an APP server.

AtomPub services are organized in a hierarchical manner:

1. Services

A grouping of workspaces.

2. Workspaces

A grouping of collections.

3. Collections

An Atom feed that contains Atom entries. Entries can be created, deleted, updated, and so on, through the HTTP methods and the mapping that AtomPub defines.

Table 7-3 summarizes these Abdera classes.

*Table 7-3 Summary of Abdera classes*

AtomPub concept	Abdera classes	Description
Service	Provider	Providers provide the implementation of an AtomPub service. The provider analyzes the request, determines what action is being requested, and forwards the request on to the appropriate method.
Workspace	WorkspaceInfo and Workspace Manager	The WorkspaceInfo class provides metadata about the workspace for the services document. The WorkspaceManager provides a way to list out the workspaces in a service.
Collection	CollectionAdapter	A CollectionAdapter is where the business logic usually lies and allows implementing the basic GET/DELETE/POST/etc. operations for an AtomPub collection.

### How an Abdera server works

The AbderaServlet is the entry point to the Abdera server framework. The Abdera servlet receives a request, creates a RequestContext object (a wrapper for the HttpServletRequest object), and forwards it on to an array of filter objects and a provider.



The `RequestContext` asks the provider for a target resolver to determine the target of the request. By default, targets have an associated `TargetType` that map to specific kinds of AtomPub artifacts (that is, collections, entries, service documents, and so on). If filters are used, they operate the same way as servlet filters (chaining filters, passing along the request) but use the Abdera APIs. These filters may intercept or alter the request.

After the request has been filtered, the provider analyzes the request. That is, is it for an Atom service document? If so, the provider uses the information supplied by the `WorkspaceInfo`, `CollectionInfo`, `CategoriesInfo`, and `CategoryInfo` interfaces.

If the request is for another type of artifact, the provider uses the workspace manager to select a collection adapter to handle the request. The `Collection` adapter is the interface that connects the AtomPub protocol to the back-end implementation. It accomplishes this by exposing methods such as `getFeed`, `getEntry`, and so on. Once the correct adapter has been selected, the provider will hand over control to the appropriate method (depending on target, type, and request method).

A target builder (supplied by the provider) can be used to construct URLs for targets (accessed through the `RequestContext`). On completing the request it passes a `ResponseContext` object back to the provider, which in turn passes it back to the Abdera servlet, which writes it out to the `HttpServletResponse` object.

In the `FeedSphere` sample application we demonstrate implementing a custom provider and `TargetResolver` to form a flexible yet powerful APP server implementation.

Visit the Abdera home page for full documentation and APIs:

<http://cwiki.apache.org/confluence/display/ABDERA/Index>

## 7.4.8 Atom Publishing Protocol support

This section demonstrates the ability to create, retrieve, update, and delete Atom content using the Atom Publishing Protocol support in Abdera.

### Reading the service document

The first step to using any APP-enabled service is to determine what collections are available and what types of resources those collections contain. The Atom protocol specification defines an XML format known as a service document that a client can use to introspect an endpoint.

To retrieve the service document, send an HTTP GET request to the service document URI. In the FeedSphere sample it is:

`http://localhost:9080/FeedSphereApp/atom/feedsphere`

The request's initial point of entry is the implementation of the `AbderaServlet`, `FeedSphereAPPServlet`. This servlet overrides the service context related method. See Example 7-24.

*Example 7-24 Extending AbderaServlet: FeedSphereAPPServlet*

---

```
public class FeedSphereAPPServlet extends AbderaServlet {

    private static final long serialVersionUID =
        -2860244280292936078L;
    private DefaultServiceContext abderaServiceContext =
        new FeedSphereServiceContext();

    protected ServiceContext createServiceContext() {
        abderaServiceContext.init(getAbdera(),
            getProperties(getServletConfig()));
        return abderaServiceContext;
    }

    protected void doGet(HttpServletRequest req,
        HttpServletResponse resp)
        throws ServletException, IOException {
        req.getRequestDispatcher("index.jsp").
            forward(req, resp);
    }

    protected void doPost(HttpServletRequest req,
        HttpServletResponse resp)
        throws ServletException, IOException {
        req.getRequestDispatcher("index.jsp").
            forward(req, resp);
    }
}
```

---

`FeedSphereServiceContext()` binds the `FeedSphere` servlet to the `FeedSphere` provider (an extension of the root `AbstractProvider` class, which handles the default request dispatching implementation) and the `FeedSphere` `targetResolver` (an extension of the `RegexTargetResolver` that uses regexes to analyze request URIs and select the appropriate target resource and type). This service context s

to be indicated to the application's servlet. Otherwise, it will pick up the default service context.

In this example, the `FeedSphereServlet` explicitly uses the `FeedSphere` service context. This could also be configured in the `Web.xml` using the `SERVICE_CONTEXT` variable to point to the fully qualified name of the service context class. See Example 7-25.

*Example 7-25 Binding the servlet to the provider manager and target resolver*

---

```
public class FeedSphereServiceContext
    extends DefaultServiceContext {

    public FeedSphereServiceContext() {
        this.defaultprovidermanager =
            FeedSphereProviderManager.class.getName();
        this.defaulttargetresolver =
            FeedSphereTargetResolver.class.getName();
    }
}
```

---

Once the servlet services the request, the target resolver determines how the provider will deal with the incoming request by pattern-matching the URI. The `FeedSphereTargetResolver` provides a mapping of URLs to service/feed/entry content. See Example 7-26.

*Example 7-26 Using the target resolver to maps URLs to content*

---

```
public class FeedSphereTargetResolver
    extends RegexTargetResolver
    implements FeedSphereConstants {

    public FeedSphereTargetResolver() {
        ResourceBundle resource = ResourceBundle
            .getBundle("com.ibm.ajax.feeds.app.samples." +
                "feedsphere_settings");

        String initialPattern =
            (String)resource.getObject("SERVLET_PATH");
        if (initialPattern == null) initialPattern = "";

        // return the service document -
        //http://server-context/feedsphere/
        setPattern(initialPattern +
            "/feedsphere(\\?[^\#]*)?",
```

```

        TargetType.TYPE_SERVICE);

    // returns the feed document -
    //http://server-context/feedsphere/feed
    setPattern(initialPattern +
        "/feedsphere/feed(\\?[^\#]*)?",
        TargetType.TYPE_COLLECTION);

    // returns a specific entry -
    //http://server-context/feedsphere/feed/1001
    setPattern(initialPattern +
        "/feedsphere/feed/([^\#?]+)(\\?[^\#]*)?",
        TargetType.TYPE_ENTRY);
    }
}

```

---

In this scenario the URL matches the service document pattern. Therefore, the request will look for a `getService` method in the provider class, as shown in Example 7-27. This class generates, maintains, and performs updates on its Atom content.

*Example 7-27 getService method of provider implementation*

---

```

public ResponseContext getService(
    RequestContext request) {
    Abdera abdera = request.getAbdera();
    Document service = getServiceDocument(abdera);
    AbstractResponseContext rc =
        new BaseResponseContext(service);
    rc.setEntityTag(service_etag);
    return rc;
}

```

---

In this method the provider gets an `Abdera` object, which is eventually passed to the `initializeServiceDocument` method where it is used to obtain a factory object. `Service`, `workspace`, and `collection` objects are created and configured (the tags and names strings are from an interface provided with the `FeedSphere` sample), and finally the service object returns its document object, as shown in Example 7-28.

*Example 7-28 Initializing the service document*

---

```

private Document initializeServiceDocument(
    Abdera abdera) {
    Factory factory = abdera.getFactory();

```

```

Service service = factory.newService();
Workspace workspace = service.addWorkspace("FeedSphere Workspace");
try {
    // Collection url is /atom/feedsphere/feed
    StringBuffer collectionURL =
        new StringBuffer(SERVICE_DOCUMENT_TAG)
            .append("/").append(FEED_TAG);
    Collection collection =
        workspace.addCollection(COLLECTION_NAME,
            collectionURL.toString());
    String[] collectionName = new String[1];
    collectionName[0] = ACCEPT_ATOM;
    collection.setAccept(collectionName);
    collection.addCategories().setFixed(false);
} catch (Exception e) {
}
return service.getDocument();
}

```

---

An `AbstractResponseContext` object is assigned to a new instance of `BaseResponseContext` provided with the document object. This object encapsulates the server's response and provides basic server response information, for example, the return code, content-type, date, and so on. Finally, the entity tag *feedsphere* is set on the `ResponseContext` object, and it is returned to be written to output.

**Note:** The ETag header field is used by HTTP/1.1 for Entity tags. Entity tags are used for comparing two or more entities from the same requested resource. An entity tag *must* be unique across all versions of all entities associated with a particular resource. A given entity tag value may be used for entities obtained by requests on different URIs without implying anything about the equivalence of those entities. Entity tags can be used in order to implement a cache system.

The cURL command-line tool showing the output of the request to the service URI is shown in Figure 7-34.

```
c:\dev\ibm>curl -v -X GET http://localhost:9080/FeedSphereApp/atom/feedsphere
* About to connect() to localhost port 9080 (#0)
* Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 9080 (#0)
> GET /FeedSphereApp/atom/feedsphere HTTP/1.1
> User-Agent: curl/7.18.1 (i386-pc-win32) libcurl/7.18.1 OpenSSL/0.9.8g zlib/1.2.3
> Host: localhost:9080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/atomsvc+xml; charset=UTF-8
< ETag: "feedsphere"
< Content-Language: en-IE
< Transfer-Encoding: chunked
< Date: Fri, 27 Jun 2008 01:41:29 GMT
< Server: WebSphere Application Server/6.1
<
<?xml version="1.0" encoding="UTF-8"?><service xmlns="http://www.w3.org/2007/app" xmlns:atom="http://www.w3.org/2005/Atom"><workspace><atom:title type="text" xmlns:atom="http://www.w3.org/2005/Atom">FeedSphere Workspace</atom:title><collection href="feedsphere/feed"><atom:title type="text" xmlns:atom="http://www.w3.org/2005/Atom">IBM User Collection</atom:title><accept>application/atom+xml;type=entry</accept><categories xmlns:atom="http://www.w3.org/2005/Atom"></categories></collection></workspace></service>* Connection #0 to host localhost left intact
* Closing connection #0
```

Figure 7-34 Retrieving the service document

The previous service document is shown in Figure 7-35 in a more readable format.

```
<?xml version="1.0" encoding="UTF-8"?>
<service xmlns="http://www.w3.org/2007/app"
  xmlns:atom="http://www.w3.org/2005/Atom">
  <workspace>
    <atom:title type="text"
      xmlns:atom="http://www.w3.org/2005/Atom">
      FeedSphere Workspace
    </atom:title>
    <collection href="feedsphere/feed">
      <atom:title type="text"
        xmlns:atom="http://www.w3.org/2005/Atom">
        IBM User Collection
      </atom:title>
      <accept>application/atom+xml;type=entry</accept>
      <categories
        xmlns:atom="http://www.w3.org/2005/Atom">
      </categories>
    </collection>
  </workspace>
</service>
```

Figure 7-35 The service document (readable)

Each collection element listed in the service document represents a container within which some piece of content can be stored. Workspace elements in the document serve only to group-related collections into logical sets (for example, a user's blog entries, uploaded files, bookmarks, and so on). Each service can be represented as a separate workspace in the service document. The collection element provides the address of the collection (the href attribute) and a listing of the types of content that can be added to a collection (identified by mime type in the accept elements).

### Reading the available feed

You can retrieve listings of a collection's member resources by issuing a GET request to the collection's URI:

<http://localhost:9080/FeedSphereApp/atom/feedsphere/feed>

Similarly to how the request is mapped to the provider's `getService` method (see "Reading the service document" on page 205), the target resolver will instruct the provider to look for the `getFeed` method, as shown in Example 7-29.

*Example 7-29 getFeed method in provider implementation*

---

```
public ResponseContext getFeed(RequestContext request) {  
    Abdera abdera = request.getAbdera();  
    Document feed = getFeedDocument(abdera);  
    AbstractResponseContext rc = new BaseResponseContext(feed);  
    rc.setEntityTag(calculateEntityTag(feed.getRoot()));  
    return rc;  
}
```

---



The `getFeedDocument` call eventually leads to the `generateDefaultFeed()` method that we used previously in Example 7-23 on page 195 to create the feed. Again, a unique entity tag is attached and the document is returned.

```
c:\dev\ibm>curl -v -X GET http://localhost:9080/FeedSphereApp/atom/feedsphere/feed
* About to connect() to localhost port 9080 (#0)
*   Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 9080 (#0)
> GET /FeedSphereApp/atom/feedsphere/feed HTTP/1.1
> User-Agent: curl/7.18.1 (i386-pc-win32) libcurl/7.18.1 OpenSSL/0.9.8g zlib/1.2.3
> Host: localhost:9080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/atom+xml; charset=UTF-8
< ETag: "db934248845189a7334fa0ca105081bf"
< Content-Language: en-IE
< Transfer-Encoding: chunked
< Date: Fri, 27 Jun 2008 19:52:13 GMT
< Server: WebSphere Application Server/6.1
<
<?xml version="1.0" encoding="UTF-8"?><feed xmlns="http://www.w3.org/2005/Atom"><title type="text">IBM alphaWorks Emerging Technologies : </title><link href="http://localhost:9080/"></link><updated>2008-06-27T02:11:29.015Z</updated><author><name>alphaWorks Author</name></author><id>http://www.alphaworks.ibm.com/</id><category term="technology-updates"></category><entry><title type="text">Real-Time Active Inference and Learning (RAIL)</title><link href="/2007/rail"></link><id>rail</id><updated>2008-06-27T02:11:29.015Z</updated><summary type="text">A Bayesian inference tool using active probing for real-time, adaptive problem diagnosis in distributed systems. (NEW: 07/17/2007 in home)</summary></entry><entry><title type="text">Web 2.0 Starter Toolkit for DB2</title><link href="/2007/web20starterkit"></link><id>web2db2</id><updated>2008-06-27T02:11:29.015Z</updated><summary type="text">A conveniently packaged set of products and technologies that enable the quick creation of DB2 Web services and feeds using PHP technology. (NEW: 07/26/2007 in home)</summary></entry></feed>* Connection #0 to host localhost left intact
* Closing connection #0
```

Figure 7-36 A feed in the collection

The response to this request will be an Atom feed document containing a set of entity elements. Each of these entity elements represent exactly one member resource in the collection, as illustrated in Figure 7-36.

```

<?xml version="1.0" encoding="UTF-8"?>
<feed xmlns="http://www.w3.org/2005/Atom">
  <title type="text">
    IBM alphaWorks Emerging Technologies : </title>
  <link href="http://localhost:8080/">
  </link>
  <updated>2008-06-27T02:11:29.015Z</updated>
  <author>
    <name>alphaWorks Author</name>
  </author>
  <id>http://www.alphaworks.ibm.com/</id>
  <category term="technology-updates"></category>
  <entry>
    <title type="text">
      Real-Time Active Inference and Learning (RAIL)
    </title>
    <link href="/2007/rail"></link>
    <id>rail</id>
    <updated>2008-06-27T02:11:29.015Z</updated>
    <summary type="text">
      A Bayesian inference tool using active
      probing for real-time, adaptive problem diagnosis in
      distributed systems. (NEW: 07/17/2007 in home)
    </summary>
  </entry>
  <entry>
    <title type="text">
      Web 2.0 Starter Toolkit for DB2
    </title>
    <link href="/2007/web20starterkit"></link>
    <id>web2db2</id>
    <updated>2008-06-27T02:11:29.015Z</updated>
    <summary type="text">A conveniently packaged set of
      products and technologies that enable the quick
      creation of DB2 Web services and feeds using PHP
      technology. (NEW: 07/26/2007 in home)
    </summary>
  </entry>
</feed>

```

Figure 7-37 An Atom feed document for an APP Collection

The entries themselves are ordered according to the value of each entry's updated element with the most recently updated entries listed first.

Additionally, the listing of entries can span multiple Atom feed documents linked to one another using paging links, as shown in Example 7-30.

*Example 7-30 Paging links in an Atom feed document*

---

```
<feed xmlns="http://www.w3.org/2005/Atom"
      xml:base="http://example.org/blog/entries?page2">
  <link rel="next" href="entries?page3" />
  <link rel="previous" href="entries?page1" />
  ...
```

---

By using links, large collection listings can be separated into smaller, more manageable documents.

### Reading a specific entry in the feed

First issue a GET request to the members' Edit URI (taken from the id element of the previous feed):

<http://localhost:9080/FeedSphereApp/atom/feedsphere/feed/rail>

The target resolver examines the URI. However, this time it results in the provider's `getEntry` method being called, as shown in Example 7-31.

*Example 7-31 getEntry*

---

```
public ResponseContext getEntry(RequestContext request) {
    Entry entry = (Entry) getAbderaEntry(request);
    if (entry != null) {
        Feed feed = (Feed) entry.getParentElement();
        entry = (Entry) entry.clone();
        entry.setSource(feed.getAsSource());
        Document entry_doc = entry.getDocument();
        AbstractResponseContext rc = new BaseResponseContext(entry_doc);
        rc.setEntityTag(calculateEntityTag(entry));
        return rc;
    } else {
        return new EmptyResponseContext(404);
    }
}
```

---

The entry object is discovered through the `getAbderaEntry` method, as shown in Example 7-32. It can also be seen that if the entry is not found, a 404 (HTTP response code representing the resource was not found) `EmptyResponseContext` object is returned instead.

*Example 7-32 `getAbderaEntry`*

---

```
private Entry getAbderaEntry(RequestContext request) {  
    Abdera abdera = request.getAbdera();  
    String entry_id = getEntryID(request);  
    Document feed = getFeedDocument(abdera);  
    try {  
        Feed currentFeed = (Feed) feed.getRoot();  
        return (Entry) currentFeed.getEntry(entry_id);  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    return null;  
}
```

---

The entry ID is parsed from the URL, and the document object (feed) provides the mechanism to find the entry object with the specified entry ID. See Figure 7-38.

```
c:\dev\ibm>curl -v -X GET http://localhost:9080/FeedSphere
App/atom/feedsphere/feed/rail
* About to connect() to localhost port 9080 (#0)
* Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 9080 (#0)
> GET /FeedSphereApp/atom/feedsphere/feed/rail HTTP/1.1
> User-Agent: curl/7.18.1 (i386-pc-win32) libcurl/7.18.1 O
penSSL/0.9.8g zlib/1.2.3
> Host: localhost:9080
> Accept: */*
>
< HTTP/1.1 200 OK
< Content-Type: application/atom+xml; type=entry; charset=
UTF-8
< ETag: "01a32891dcd2df9c46bba9a0edfa3695"
< Content-Language: en-IE
< Transfer-Encoding: chunked
< Date: Mon, 30 Jun 2008 04:56:27 GMT
< Server: WebSphere Application Server/6.1
<
<?xml version="1.0" encoding="UTF-8"?><entry xmlns="http:/
/www.w3.org/2005/Atom"><title type="text">Real-Time Active
Inference and Learning (RAIL)</title><link href="/2007/ra
il"></link><id>rail</id><updated>2008-06-30T04:43:18.875Z<
/updated><summary type="text">A Bayesian inference tool us
ing active probing for real-time, adaptive problem diagnos
is in distributed systems. (NEW: 07/17/2007 in home)</summ
ary><source><title type="text">IBM alphaWorks Emerging Te
chnologies : </title><link href="http://localhost:8080/"><
/link><updated>2008-06-30T04:43:18.859Z</updated><author><
name>alphaWorks Author</name></author><id>http://www.alpha
works.ibm.com/</id><category term="technology-updates"></c
ategory></source></entry>* Connection #0 to host localhost
left intact
* Closing connection #0
```

Figure 7-38 An Atom entry document representing a resource

```

<?xml version="1.0" encoding="UTF-8"?>
<entry xmlns="http://www.w3.org/2005/Atom">
  <title type="text">
    Real-Time Active Inference and Learning (RAIL)
  </title>
  <link href="/2007/rail"></link>
  <id>rail</id>
  <updated>2008-06-30T04:43:18.875Z</updated>
  <summary type="text">
    A Bayesian inference tool using active probing for
    real-time, adaptive problem diagnosis in distributed
    systems. (NEW: 07/17/2007 in home)
  </summary>
  <source>
    <title type="text">
      IBM alphaWorks Emerging Technologies :
    </title>
    <link href="http://localhost:8080/"></link>
    <updated>2008-06-30T04:43:18.859Z</updated>
    <author><name>alphaWorks Author</name></author>
    <id>http://www.alphaworks.ibm.com/</id>
    <category term="technology-updates"></category>
  </source>
</entry>

```

Figure 7-39 An entry of the feed

**Tip:** Use the optional If-Match and If-Unmodified-Since headers in a PUT request to allow APP implementations to protect against overwriting modifications that other clients might have made on a member resource. If either of these conditions is not met, the server should reject the request and notify the client that it is likely that there was a conflict with the resource that they are attempting to modify. If the conditions are met and the server considers the modifications submitted by the client to be acceptable, it will respond with an appropriate success response.

## Creating a new entry

The new entry.xml file created is shown in Figure 7-40.

```
<entry xmlns="http://www.w3.org/2005/Atom">
  <title>A New Entry</title>
  <link href="http://www.example.org/blog/sample"></link>
  <description>Sample Text</description>
  <id>sample</id>
  <updated>2008-06-30T05:20:43.375Z</updated>
</entry>
```

Figure 7-40 entry.xml

If the URL matches the entry pattern for the target resolver and the request is of type POST, Abdera directs the call to the provider's createEntry method, as shown in Example 7-33.

Example 7-33 Creating an entry

```
public ResponseContext createEntry(RequestContext request) {
    Abdera abdera = request.getAbdera();
    try {
        Document entry_doc = constructEntryDocument(request, abdera);
        if (entry_doc != null) {
            Entry entry = (Entry) entry_doc.getRoot();
            manageAnEntry(entry, null, abdera);
            BaseResponseContext rc = new BaseResponseContext(entry);
            IRI baseUri = resolveBase(request);
            rc.setLocation(
                baseUri.resolve(entry.getEditLinkResolvedHref())
                    .toString());
            rc.setContentLocation(rc.getLocation().toString());
            rc.setEntityTag(calculateEntityTag(entry));
            rc.setStatus(201);
            return rc;
        } else {
            return new EmptyResponseContext(400);
        }
    } catch (ParseException pe) {
        return new EmptyResponseContext(415);
    } catch (ClassCastException cce) {
        return new EmptyResponseContext(415);
    }
}
```

```

    } catch (Exception e) {
        return new EmptyResponseContext(400);
    }
}

```

---

The `constructEntryDocument` method creates a skeletal Atom document object for the entry to be later populated and returned, as shown in Example 7-34.

*Example 7-34 Creating a skeletal entry document*

---

```

private Document constructEntryDocument(RequestContext request,
    Abdera abdera) throws Exception {
    MimeType contentType = request.getContentType();
    if (contentType != null
        && !MimeTypeHelper.isAtom(contentType.toString()))
        throw new ParseException();
    Document entry_doc =
        (Document) request.getDocument(abdera.getParser())
            .clone();
    return entry_doc;
}

```

---

The `manageAnEntry` method is a utility method to manage the creation or update of an entry. It uses Abdera's internationalization (i18n) package to provide an international-friendly resource identifier for the new entry. It sets the entry's last updated time (now), link value, and any edit URI applicable. See Example 7-35.

*Example 7-35 Configuring the new entry*

---

```

private void manageAnEntry(Entry currentEntry, Entry originalEntry,
    Abdera abdera) throws Exception {
    IRI id = new IRI(abdera.getFactory().newUuidUri());
    if (originalEntry != null) {
        id = originalEntry.getId();
        if (!currentEntry.getId().equals(id))
            throw new Exception("409");
        originalEntry.discard();
    }
    currentEntry.setUpdated(new Date());
    currentEntry.getIdElement().setValue(id.toString());
    StringBuffer collectionURL =
        new StringBuffer(SERVICE_DOCUMENT_TAG)
            .append("/").append(FEED_TAG).append("/");
    currentEntry.addLink(collectionURL.append(
        currentEntry.getId().toString()).toString(), "edit");
}

```



```

Feed feed = (Feed) getFeedDocument(abdera).getRoot();
feed.insertEntry(currentEntry);
feed.setUpdated(new Date());
}

```

createEntry ends by setting the responseContext object's response header fields with location, status, and entity tag data. Exceptions are caught and associated with HTTP response codes. 400 indicates a bad request and 415 corresponds to unsupported media type (that is, incorrect Atom protocol format). Should all go well, the entry is created and a HTTP response code of 201 is returned.

Here we use cURL command-line tool to post the entry contained in entry.xml to the server. The results are shown in Figure 7-41.

```

c:\dev\ibm>curl -v -X POST -H "Content-Type: application/atom+xml" --data @entry.xml http://localhost:9080/FeedSphereApp/atom/feedsphere/feed
* About to connect() to localhost port 9080 (#0)
* Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 9080 (#0)
> POST /FeedSphereApp/atom/feedsphere/feed HTTP/1.1
> User-Agent: curl/7.18.1 (i386-pc-win32) libcurl/7.18.1 OpenSSL/0.9.8g zlib/1.2.3
> Host: localhost:9080
> Accept: */*
> Content-Type: application/atom+xml
> Content-Length: 227
>
< HTTP/1.1 201 Created
< Content-Type: application/atom+xml; type=entry; charset=UTF-8
< Content-Location: http://localhost:9080/FeedSphereApp/atom/feedsphere/feedsphere/feed/urn:uuid:FD2BC387AE3B5D7AAD1214803851696
< ETag: "b12fdedc37dd02c8b3dbcb4874bd2cc7"
< Location: http://localhost:9080/FeedSphereApp/atom/feedsphere/feedsphere/feed/urn:uuid:FD2BC387AE3B5D7AAD1214803851696
< Content-Language: en-IE
< Transfer-Encoding: chunked
< Date: Mon, 30 Jun 2008 05:30:50 GMT
< Server: WebSphere Application Server/6.1
<
<entry xmlns="http://www.w3.org/2005/Atom"><title type="text">A New Entry</title><link href="http://www.example.org/blog/sample"></link><description>Sample Text</description><id>urn:uuid:FD2BC387AE3B5D7AAD1214803851696</id><updated>2008-06-30T05:30:51.692Z</updated><link href="feedsphere/feed/urn:uuid:FD2BC387AE3B5D7AAD1214803851696" rel="edit"></link></entry>* Connection #0 to host localhost left intact

* Closing connection #0

```

Figure 7-41 The new entry POSTed to the server

Note the HTTP response code of 201 Created. The server also returns the new entry and its location details. Also note that the ID was automatically created and assigned. The server implementation in the FeedSphereApp sample does this to guarantee a unique ID for the new entry.

## Editing an entry

For an entry to be editable, the `<link rel="edit" ... />` tag must be present. For example:

```
<link rel="edit" href=
"http://localhost:9080/FeedSphereApp/atom/feedsphere/feed/sample"/>
```

Entries can then be edited by PUTting to the entry's edit URI. For example:

```
curl -v -X PUT -H "Content-Type: application/atom+xml" --data
@entry.xml
http://localhost:9080/FeedSphereApp/atom/feedsphere/feed/sample
```

The `updateEntry` method shown in Example 7-36 handles altering the entries.

### *Example 7-36 Updating/editing an existing entry*

---

```
public ResponseContext updateEntry(RequestContext request) {
    Abdera abdera = request.getAbdera();
    Entry orig_entry = getAbderaEntry(request);
    if (orig_entry != null) {
        try {
            Document entry_doc = constructEntryDocument(request, abdera);
            if (entry_doc != null) {
                Entry entry = (Entry) entry_doc.getRoot();
                manageAnEntry(entry, orig_entry, abdera);
                return new EmptyResponseContext(204);
            } else {
                return new EmptyResponseContext(400);
            }
        } catch (ParseException pe) {
            return new EmptyResponseContext(415);
        } catch (ClassCastException cce) {
            return new EmptyResponseContext(415);
        } catch (Exception e) {
            if ("409".equalsIgnoreCase(e.getMessage()))
                return new EmptyResponseContext(409);
            return new EmptyResponseContext(400);
        }
    }
}
```

```

    } else {
        return new EmptyResponseContext(404);
    }
}

```

---

We have seen before how an existing entry can be retrieved (Example 7-32 on page 216) and how a skeletal entry is constructed (Example 7-34 on page 220). An update entry basically recreates the entry with the existing entry data (manageAnEntry is passed to the original entry now instead of null). As shown in Example 7-35 on page 220, a 409 (conflict) failure response code is returned if the entry IDs do not match.

If successful, editing an entry should return with the 204 HTTP status code.

The FeedSphereApp server implementation creates its own unique IDs (overwriting the supplied ID). The edit URI is not known before entry creation. We will alter entry.xml by providing new description text and replacing any reference to the old ID (sample) with the new one (urn:uuid:FD2BC387AE3B5D7AAD1214826723788).

```

<entry xmlns="http://www.w3.org/2005/Atom">
  <title>A New Entry</title>
  <link>
    href="http://localhost:9080/FeedSphereApp/atom/feedsphere/feed/urn:uuid:FD2BC387AE3B5D7AAD1214826723788"</link>
    <link rel="edit" href="
"http://localhost:9080/FeedSphereApp/atom/feedsphere/feed/urn:uuid:FD2BC387AE3B5D7AAD1214826723788" />
  <description>Even Newer Sample Text</description>
  <id>urn:uuid:FD2BC387AE3B5D7AAD1214826723788</id>
  <updated>2008-06-30T05:20:43.375Z</updated>
</entry>

```

Figure 7-42 Modified entry.xml

Figure 7-43 shows the cURL command-line tool output.

```
c:\dev\ibm>curl -v -X PUT -H "Content-Type: application/atom+xml" --data @entry.xml http://localhost:9080/FeedSphereApp/atom/feedsphere/feed/urn:uuid:FD2BC387AE3B5D7AAD1214826723788
* About to connect() to localhost port 9080 (#0)
* Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 9080 (#0)
> PUT /FeedSphereApp/atom/feedsphere/feed/urn:uuid:FD2BC387AE3B5D7AAD1214826723788 HTTP/1.1
> User-Agent: curl/7.18.1 (i386-pc-win32) libcurl/7.18.1 OpenSSL/0.9.8g zlib/1.2.3
> Host: localhost:9080
> Accept: */*
> Content-Type: application/atom+xml
> Content-Length: 459
>
< HTTP/1.1 204 No Content
< Content-Language: en-IE
< Content-Length: 0
< Date: Mon, 30 Jun 2008 12:27:13 GMT
< Server: WebSphere Application Server/6.1
<
* Connection #0 to host localhost left intact
* Closing connection #0
```

Figure 7-43 cURL edit entry output

## Deleting an entry

To remove a resource from a collection, send a DELETE request to the entry's edit URI.

The deleteEntry method shown in Example 7-37 is very simple. It attempts to find the entry of the DELETE request references, calls the discard() method on the entry object, and returns a HTTP response code of 204 (no content) indicating that the deletion was performed successfully.

Example 7-37 deleteEntry

---

```
public ResponseContext deleteEntry(RequestContext request) {
    Entry entry = getAbderaEntry(request);
    if (entry != null)
        entry.discard();
    return new EmptyResponseContext(204);
}
```

---

For instance, to remove the new entry created in Figure 7-41 on page 221, issue the DELETE request shown in Figure 7-44.

```
c:\dev\ibm>curl -v -X DELETE http://localhost:9080/FeedSphereApp/atom/feedsphere/feed/urn:uuid:FD2BC387AE3B5D7AAD1214803851696
* About to connect() to localhost port 9080 (#0)
* Trying 127.0.0.1... connected
* Connected to localhost (127.0.0.1) port 9080 (#0)
> DELETE /FeedSphereApp/atom/feedsphere/feed/urn:uuid:FD2BC387AE3B5D7AAD1214803851696 HTTP/1.1
> User-Agent: curl/7.18.1 (i386-pc-win32) libcurl/7.18.1 OpenSSL/0.9.8g zlib/1.2.3
> Host: localhost:9080
> Accept: */*
>
< HTTP/1.1 204 No Content
< Content-Language: en-IE
< Content-Length: 0
< Date: Mon, 30 Jun 2008 06:22:17 GMT
< Server: WebSphere Application Server/6.1
<
* Connection #0 to host localhost left intact
* Closing connection #0
```

Figure 7-44 Deleting an entry

Following a successful delete, the 204 HTTP status code is returned and the entry no longer appears within the collections Atom feed.

## 7.4.9 Abdera APP client

An `AbderaClient` implementation can be used to programmatically interact with the AtomPub server.

### Creating `AbderaClient` instance

The code snippet shown in Example 7-38 can be used to create an `AbderaClient` instance.

Example 7-38 Creating an `AbderaClient` instance

---

```
Abdera abdera = new Abdera();
AbderaClient client = new AbderaClient(abdera);
```

---

### PUTting resources

PUTting using the `AbderaClient` is relatively simple. Once an object is created that inherits from the `model.Base` class (in this case, `Entry`) to be PUT, call the `put` method on the client and pass it the URI, entry, and any `RequestOptions`

(that is, content headers) desired. The `RequestOptions` class provides access to all HTTP response headers. See Example 7-39.

*Example 7-39 PUTting to FeedSphere*

---

```
String uri = constructBaseURI().toString() + "/Web2db2";
Entry entry = factory.newEntry();
entry.setTitle("[Updated] Web 2.0 Starter Toolkit for DB2");
entry.setId("Web2db2");
entry.addAuthor("IBM User 1");
...
ClientResponse res = null;
RequestOptions opts = new RequestOptions();
opts.setContentType("application/atom+xml;type=entry");
if (PUT.equalsIgnoreCase(command))
    res = client.put(accessURI, entry, opts);
...
log(command + " status: " + res.getStatus());
```

---

**Note:** Non-Abdera resources can also be used:

```
InputStream in = ...
InputStreamRequestEntity entity =
    new InputStreamRequestEntity(in, "image/png");
ClientResponse resp =
    client.put("http://www.example.org/collection", entity);
```

## GETting resources

Example 7-40 demonstrates a call to the `AbderaClient` instance's `get` method.

*Example 7-40 GETting from FeedSphere*

---

```
String uri = constructBaseURI().toString() + "/Web2db2";
...
ClientResponse res = null;
RequestOptions opts = new RequestOptions();
if (GET.equalsIgnoreCase(command))
    res = client.get(accessURI, opts);
...
log(command + " status: " + res.getStatus());
```

---

## POSTing resources

POSTing a resource is very similar to PUTting. See Example 7-41.

*Example 7-41 POSTing to FeedSphere*

---

```
String uri = constructBaseURI().toString();
Entry entry = factory.newEntry();
entry.setTitle("Web Browser based interaction with the Eclipse IDE");
entry.setUpdated(new Date());
entry.addAuthor("IBM");
entry.setId("eclifox");
entry.setSummary("A new technology that demonstrates Web based access
to Eclipse");
...
ClientResponse res = null;
RequestOptions opts = new RequestOptions();
if (POST.equalsIgnoreCase(command))
    res = client.post(accessURI, entry, opts);
...
log(command + " status: " + res.getStatus());
```

---

## Deleting resources

It is a good practice to ensure that the resource to be deleted exists before attempting the DELETE by first performing a GET command and ensuring that the HTTP response code is 200, as shown in Example 7-42.

*Example 7-42 Deleting from FeedSphere*

---

```
String uri = constructBaseURI().toString() + "/rail";
ClientResponse res = client.get(uri);
if (res.getStatus() != 200)
    // do not continue with the delete
ClientResponse res = null;
RequestOptions opts = new RequestOptions();
if (DELETE.equalsIgnoreCase(command))
    res = client.delete(accessURI, opts);
...
log(command + " status: " + res.getStatus());
```

---

## Using custom HTTP methods

To use custom HTTP methods:

```
client.execute("PATCH", "http://example.org/foo", entity, null);
if (resp.getType() == ResponseType.SUCCESS) {
    // success
} else {
    // there was an error
}
```

## Using SSL

To use Abdera to access SSL-protected endpoints, you must register a trust manager, as shown in Example 7-43. Abdera ships with a default non-op Trust Manager implementation that is designed to make it possible to use SSL services without providing any level of trust verification.

*Example 7-43 Registering a trust manager*

---

```
Abdera abdera = new Abdera();
AbderaClient client = new AbderaClient(abdera);

// Default trust manager provider registered for port 443
AbderaClient.registerTrustManager();

client.get("https://localhost:9080/foo");
```

---

TrustManager will be registered on the default SSL port 443. Should a different port be required, pass the port to the TrustManager as shown below:

```
AbderaClient.registerTrustManager(1234);
```

Alternatively, you can implement your own TrustManager.



You can also use SSL-based Client Certificate Authentication, as shown in Example 7-44.

*Example 7-44 Using SSL-based Client Certificate Authentication*

---

```
KeyStore keystore = null;
ClientAuthSSLProtocolSocketFactory factory =
    new ClientAuthSSLProtocolSocketFactory(
        keystore, "keystorepassword");

AbderaClient.registerFactory(factory, 443);

// DO NOT register a trust manager after this point

client.get("https://localhost:9080/foo");
```

---

## Authentication

Abdera can use HTTP authentication when requesting resources. See Example 7-45.

*Example 7-45 Adding basic authentication*

---

```
client.addCredentials(
    "http://example.org",
    "realm",
    "basic",
    new UsernamePasswordCredentials(
        "username",
        "password")
);
```

---

When basic authentication is required at the specified URL (<http://example.org>), the above user name and password will be supplied.

Custom authentication may be used by setting the authorization header explicitly:

```
RequestOptions options = client.getDefaultRequestOptions();
options.setAuthorization("MyCustomAuthenticationSystem");
```

## Cookies and caching

Abdera does not currently have an API for working with cookies, but the underlying HTTP client implementation (Apache HttpClient) supports and uses cookies within a single session.

A built-in HTTP client cache support is supplied with the Abdera client for the HTTP cache-control mechanism. The default cache implementation is an in-memory LRU cache that will not persist cached resources permanently. All of the standard cache-control mechanisms are supported (except vary). When a GET request is issued multiple times by the client on a cached resource, the cached copy will be returned if it is not considered stale.

To disable the client cache:

```
RequestOptions options = client.getDefaultRequestOptions();  
options.setUseLocalCache(false);
```

To set HTTP cache-invalidation properties:

```
RequestOptions options = client.getDefaultRequestOptions();  
options.setNoCache(true);  
  
options.setMaxAge(10);
```

# REST

This chapter presents the core concepts of the REST architecture and how these can be applied to the WebSphere Application Server solutions using the Web 2.0 Feature Pack.

This chapter discusses the following topics:

- ▶ Introduction to REST
- ▶ Building a REST Servlet
- ▶ REST-as-CRUD or REST-as-protocol
- ▶ REST or SOA: Making the decision
- ▶ REST and Ajax
- ▶ REST in the industry

## 8.1 Introduction to REST

Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web. The terms *Representational State Transfer* and *REST* were introduced in 2000 in the doctoral dissertation of Roy Fielding, one of the principal authors of the Hypertext Transfer Protocol (HTTP) specification. The terms have since come into widespread use in the networking community.

REST strictly refers to a collection of network architecture principles that outline how resources are defined and addressed. The term is often used in a looser sense to describe any simple interface that transmits domain-specific data over HTTP without an additional messaging layer such as SOAP or session tracking via HTTP cookies.

**Note:** Systems that follow Fielding's REST principles are often referred to as RESTful.

### Principles and benefits

REST is receiving significant interest from the next generation of Internet architects because of its ability to handle the Web's scalability and growth requirements in a simple and easy to understand way. This is a direct result of a few key design principles:

- ▶ Application state and functionality are divided into resources.
- ▶ Every resource is uniquely addressable using a universal syntax for use in hypermedia links.
- ▶ All resources share a uniform interface for the transfer of state between client and resource, consisting of:
  - A constrained set of well-defined operations
  - A constrained set of content types, optionally supporting code on demand
- ▶ Internet browsers have the mechanics to invoke and consume asynchronous REST requests with no additional coding.
- ▶ REST payloads can contain different MIME types ranging from XML to JSON to TEXT.
- ▶ REST offers an ability to move back to simplicity rather than being drowned in the world of enterprise standards.
- ▶ Provides improved response times and server loading characteristics due to support for caching.

- ▶ Improves server scalability by reducing the need to maintain communication state. This means that different servers can be used to handle initial and subsequent requests.
- ▶ Requires less client-side software to be written than other approaches because a single browser can access any application and any resource.
- ▶ Depends less on vendor software than mechanisms that layer additional messaging frameworks on top of HTTP.
- ▶ Provides equivalent functionality when compared to alternative approaches to communication.
- ▶ Does not require a separate resource discovery mechanism due to the use of hyperlinks in content.
- ▶ Provides better long-term compatibility and evolvability characteristics than RPC due to:
  - The capability of document types such as HTML to evolve without breaking backwards-compatibility or forwards-compatibility
  - The ability of resources to add support for new content types as they are defined without dropping or reducing support for older content types

The REST client-server separation of concerns simplifies component implementation, reduces the complexity of connector semantics, improves the effectiveness of performance tuning, and increases the scalability of pure server components. Layered system constraints allow intermediaries—proxies, gateways, and firewalls—to be introduced at various points in the communication without changing the interfaces between components, thus allowing them to assist in communication translation or improve performance via large-scale, shared caching. REST enables intermediate processing by constraining messages to be self-descriptive:

- ▶ Interaction is stateless between requests.
- ▶ Standard methods and media types are used to indicate semantics and exchange information.
- ▶ Responses explicitly indicate cacheability.

An important concept in REST is the existence of resources (sources of specific information), each of which can be referred to using a global identifier (a URI). In order to manipulate these resources, components of the network (clients and servers) communicate via a standardized interface (for example, HTTP) and exchange representations of these resources (the actual documents conveying the information).

## Resources are simple to implement and understand

Creating a RESTful Web service is like forming a sentence—you need a noun, a verb, and an adjective. In REST terms, nouns are the resources pointed to by URIs. The verbs are the HTTP actions on resources. The adjectives are the formats of data that the resources are represented in. A RESTful Web service can be described similarly to the way that sentences are diagrammed. For example, in Table 8-1 we show how a set of services related to a photo album application would be described.

Table 8-1 Photo Album example usage of REST

Sentence (resource description)	Noun (URI)	Verb (action)	Adjectives (formats)
List all the albums.	.../album	GET	JSON
Show JPEG photo.	.../photo	GET	JPEG
Add a JPEG photo.	.../photo	POST	JPEG
Delete a JPEG photo.	.../photo	DELETE	JPEG
List all the JPEG photos.	.../photo	GET	JPEG

Understanding Table 8-1 is easy. Each column breaks down into a different part of a sentence.

- ▶ Nouns/URIs: URLs are the most identifiable part of the Web, and as such are a straightforward way to organize your services. Organizing a unique URI for each resource avoids confusion and promotes scalability.
- ▶ Verbs/actions: You can perform exactly four HTTP operations on these URLs: POST, GET, PUT, and DELETE. (HTTP supports a few more actions, but these are the interesting operations for now.) While only having four operations available is constraining, the simplicity is somewhat liberating. The four HTTP operations roughly map to create, read, update, and delete (CRUD). CRUD provides the foundational functions needed to implement a relational database, so while simple, these four methods can be used in interesting and powerful ways.
- ▶ Adjectives/data formats: There are well-known data types (MIME types, for example, text/html, image/jpeg) that are natively supported by HTTP servers and browsers. Simple XML and JSON allow more custom data formats that are both self-describing and can easily be parsed by the user. (When we say parse we also mean read with your eyes and parse with your brain.)

Any number of connectors (for example, clients, servers, caches, tunnels, and so on) can mediate the request, but each does so without seeing past its own request (referred to as *layering*, another constraint of REST and a common principle in many other parts of information and networking architecture). Thus, an application can interact with a resource by knowing two things: the identifier of the resource and the action required. It does not need to know whether there are caches, proxies, gateways, firewalls, tunnels, or anything else between it and the server actually holding the information. The application does, however, need to understand the format of the information (representation) returned, which is typically an HTML or XML document of some kind, although it may be an image or any other content.

The other key point to make about the REST architecture is that it is not just for server-to-browser solutions. It is more than capable of being used in the server-to-server space where Web services has become common. This will allow aggregation solutions or RESTful servers to add value to business information as it flows around an organization. It could even be used at edge of network points to provide added value as the data is coming into or out of an enterprise.

## 8.2 Building a REST servlet

When J2EE was introduced the core concepts of REST and RESTful were not introduced into the core programming model. The good news is that a very flexible API was introduced that allowed for the creation of flexible components that fit into the model of the Web. The servlet API is perfect for helping model out a RESTful architecture. This means that REST can be introduced into the assets that have already been deployed.

### Design

One of the elements that is missing is a simple-to-use servlet that can be mapped to REST resource names. These resource names can then be modeled into your application data logic or business logic. This section outlines how to create these types of servlet and how they can be used in your application. It is also important to point out that at the beginning of each Web 2.0 application you need to make some design decisions about how you plan to release feeds of data into your application and to the wider community. This could effect the way in which you implement these resources. It is worth investigating whether the use of the Web 2.0 Feature Pack RPC Adapter is the best starting point. That allows you to expose your existing J2EE resource in the simplest and easiest way possible. However, if you are working towards a true RESTful architecture you may need to have more programming flexibility at your disposal. This would be a good starting point to use the REST servlet explained below.

The background for this REST servlet approach is derived from two or three successful engagements with customers where the Web 2.0 Feature Pack was deployed into WebSphere applications. The first design point was how to retrieve the data from the server using a RESTful notation and also have a module approach to building the interface points to the business logic and data.

The result was a easy-to-use servlet that could be easily configured to invoked REST Handlers that could encapsulate the core functionality of the REST resource that is being invoked.

## Implementation

To create a simple RESTful framework for use with a J2EE Dynamic Web Application follow the instructions listed below. They will lead you through the steps necessary for creating a Web project and then adding the necessary servlet code and configuration XML files. This enables you to start with a basic design that could be extended to support the specific application needs.

### Entity support

The example servlet will also support what is know as entity support. This enables you to perform entity type operations on the defined resources. You can then access individual index elements, a range of elements, or access the elements by specifying a key. Table 8-2 provides examples of what is meant by RESTful entity support.

Table 8-2 RESTful entity support in the REST servlet sample

Operation	URI pattern	HTTP operation
Retrieve an <i>employee</i> with a key of perrins.	/employees/perrins	GET
Retrieve first employee.	/employees/1	GET
Retrieve first three employees.	/employees/1..3	GET
Retrieve a range of employees.	/employees/1,3,4	POST
Delete the fourth employee.	/employees/4	DELETE

These examples offer a wide variety of flexibility for the application developer in both consuming the REST service and invoking it.



## Building a REST servlet

To build a sample REST servlet implementation that can be used in your WebSphere Application Server:

1. Create a Dynamic Web Application within the Rational Application Development tooling.
2. Select **File** → **New** → **Web** → **Dynamic Web Project**.
3. Create the project with a file name of ITSORESTProvider and an EAR file of ITSORestProviderEAR.
4. Within the src directory create the following packages:
  - com.ibm.itso.rest
  - com.ibm.itso.rest.handlers.demo
  - com.ibm.itso.rest.servlet
5. The basic principle of the REST servlet allows Web developers to build REST Handlers that implement a simple REST Handler interface. The next step is to create the basic interface implementation. Create a new class file named RestHandler in the com.ibm.itso.rest package and add the code shown in Example 8-1.

*Example 8-1 RestHandler.java interface definition*

---

```
package com.ibm.itso.rest;

import java.io.InputStream;
import java.io.OutputStream;
import java.util.HashMap;

/**
 * Simple handler interface for performing RESTful operations on a
 * resource. Takes an InputStream and set of
 * parameters as an input and supplies an OutputStream for handlers to
 * issue a response from the invoker.
 */
public interface RestHandler {

    public static final int OPERATION_CREATE = 1;
    public static final int OPERATION_READ = 2;
    public static final int OPERATION_UPDATE = 3;
    public static final int OPERATION_DELETE = 4;

    /**
     * Lifecycle method for initialising the handler instance.
     */
    public void init(HashMap params) throws RestException;
```

```

/**
 * Handles the access to a resource based on the provided I/O
 streams and parameters.
 */
public void handleOperation(int operation, String path, HashMap
parameters, InputStream inStream, OutputStream outStream) throws
RestException;

/**
 * Lifecycle method for destroying the handler instance.
 */
public void destroy() throws RestException;

}

```

6. The REST Handler supports a set of operations that can be performed on it. These operations will map directly to the RESTful principles previously presented in regard to the use of the GET, PUT, POST, and DELETE operations for HTTP and how these can be mapped to create, read, update, and delete operations.

*Table 8-3 Mapping of REST HTTP operations to RESTHandler operations*

Action	HTTP	Operation
Read from a resource URI.	GET	OPERATION_READ
Create a new item on a resource URI.	PUT	OPERATION_CREATE
Update a item on a resource URI.	POST	OPERATION_UPDATE
Delete an item from a resource URI.	DELETE	OPERATION_DELETE

7. The RestHandler class has a dependency on another class that needs to be created in the com.ibm.itso.rest package. Use the code shown in Example 8-2 to create the RestException class.

*Example 8-2 RestException class*

```

package com.ibm.itso.rest;

public class RestException extends Exception {

    private static final long serialVersionUID = 100L;

```

```
public RestException() {
    super();
}

public RestException(String message) {
    super(message);
}

public RestException(Throwable cause) {
    super(cause);
}

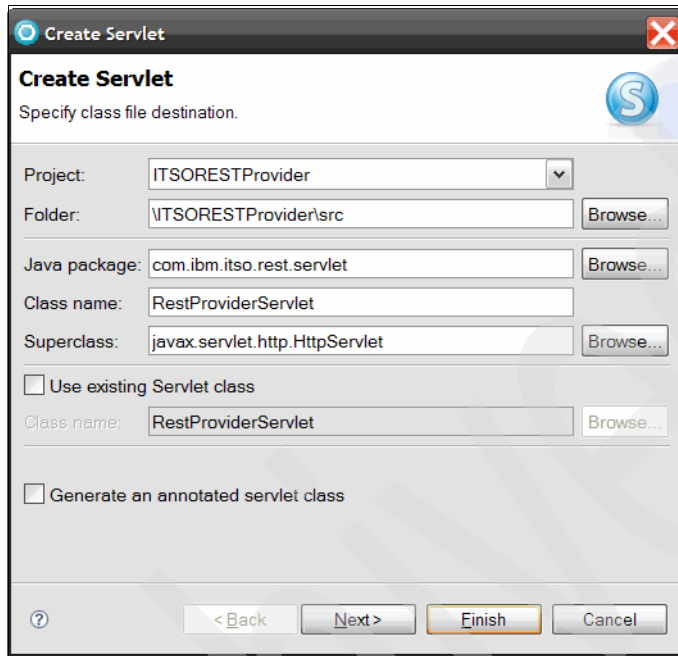
public RestException(String message, Throwable cause) {
    super(message, cause);
}
}
```

---

The REST servlet will have its meta data stored in an XML file that can be stored in the WEB-INF directory. This file allows the meta data for the REST services to be defined. The REST servlet loads this data when it is initialized. When a request is received by the servlet it attempts to match the request type against the REST services defined, and if a match is found it triggers the invocation of the REST Handler.

The next step is to create a servlet that can support the required behavior.

8. Open the Web application deployment descriptor and create a new servlet using the **Add** button. See Figure 8-1. Click **Next**.



The image shows a 'Create Servlet' dialog box with the following fields and options:

- Project:** ITSORESTProvider
- Folder:** ITSORESTProvider\src
- Java package:** com.ibm.itso.rest.servlet
- Class name:** RestProviderServlet
- Superclass:** javax.servlet.http.HttpServlet
- ☐ Use existing Servlet class
- Class name:** RestProviderServlet
- ☐ Generate an annotated servlet class

At the bottom, there are navigation buttons: < Back, Next >, Finish (highlighted), and Cancel.

Figure 8-1 Create the RestProviderServlet

9. Edit the URL mapping to match `/rest/*`. See Figure 8-2.

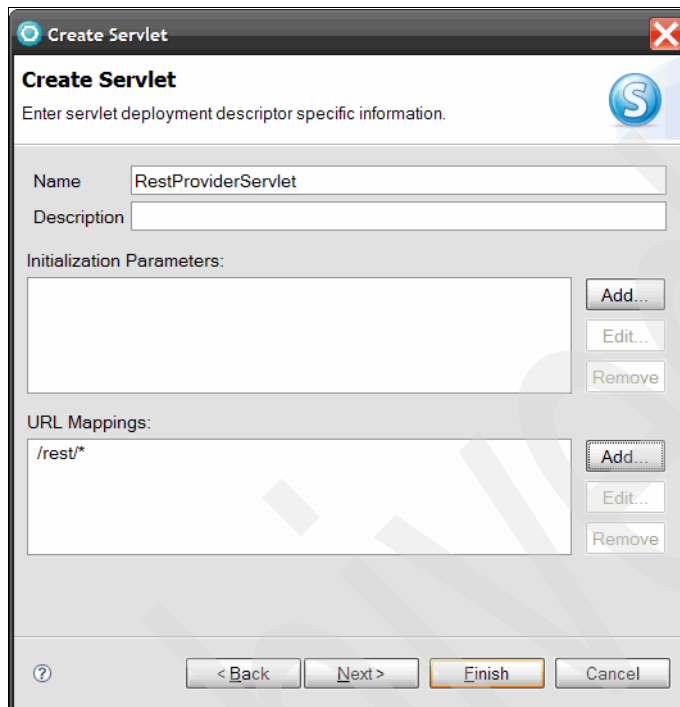


Figure 8-2 URL Mapping for `RestProviderServlet`

10. Click **Finish** to create the skeleton servlet.

11. Before you can complete the servlet you need to implement a utility class that will help with the reading of an XML configuration file. Create a class named `XMLUtil.java` in the `com.ibm.itso.rest.servlet` package. Use the code shown in Example 8-3 to complete the class.

*Example 8-3* `XMLUtil.java`

```
package com.ibm.itso.rest.servlet;

import java.io.File;
import java.io.FileOutputStream;
import java.io.OutputStreamWriter;
import java.io.StringWriter;

import org.apache.xerces.parsers.DOMParser;
import org.apache.xml.serialize.OutputFormat;
import org.apache.xml.serialize.XMLSerializer;
```

```

import org.w3c.dom.Document;
import org.xml.sax.InputSource;

/**
 * XML-related utilities.
 * @author Matt Perrins
 */
public class XMLUtil {

    public static void saveXMLDocument(String filename, Document doc)
    throws java.io.IOException{
        StringWriter sw = new StringWriter();
        OutputFormat format = new OutputFormat(doc);
        format.setIndenting(true);
        XMLSerializer serial = new XMLSerializer(sw,format);
        serial.serialize(doc);

        File newFile = new File(filename);
        FileOutputStream output = new FileOutputStream(newFile);
        OutputStreamWriter writer = new OutputStreamWriter(output);
        writer.write(sw.toString(), 0, sw.toString().length());
        writer.close();
        output.close();
    }

    public static Document parseToDocument(InputSource iSrc) throws
    org.xml.sax.SAXException, java.io.IOException{
        DOMParser dp = new DOMParser();

        dp.setFeature("http://apache.org/xml/features/nonvalidating/load-extern
        al-dtd", false);
        dp.parse(iSrc);

        Document doc = dp.getDocument();
        return doc;
    }

    public static String getXMLAsString(Document xml) throws Exception
    {
        OutputFormat format = new OutputFormat(xml);
        format.setIndenting(true);
        StringWriter stringOut = new StringWriter();
        XMLSerializer serial = new XMLSerializer( stringOut, format );
        try
        {
            serial.asDOMSerializer();

```

```

        serial.serialize( xml.getDocumentElement() );
    }
    catch (Exception e)
    {
        String errmsg = "Error outputting DOM to XML.  Extend error
message was:/n";
        throw new Exception(errmsg + e);
    }
    return stringOut.toString();
}
}

```

---

12. The next step is to implement the functionality of the servlet. The first methods implemented are the `init()` and `destroy()` methods for the servlet. Add the code shown in Example 8-4 to the servlet body, removing any existing `init()` and `destroy()` methods.

*Example 8-4 RestProviderServlet init() and destroy() methods*

---

```

private static final long serialVersionUID = 100L;

private Hashtable handlers = null;

public RestProviderServlet() {
    super();
    handlers = new Hashtable();
}

public void init(ServletConfig config) throws ServletException {
    // Parse the rest-handlers.xml file
    ServletContext sc = config.getServletContext();
    InputStreamReader reader = new
InputStreamReader(sc.getResourceAsStream("/WEB-INF/rest-handlers.xml"))
;
    try {
        Document xml = XMLUtil.parseToDocument(new
InputSource(reader));
        // Now pull out each definition from the file, create a class
and put it in the registry
        // for consumption at runtime.
        NodeList nl = xml.getElementsByTagName("handler");
        for (int i=0; i< nl.getLength(); i++) {
            Element handlerElem = (Element) nl.item(i);
            String path = handlerElem.getAttribute("path");

```

```

        if (path == null) {
            throw new ServletException("Missing path attribute on
handler element.");
        }
        String impl = handlerElem.getAttribute("impl");
        if (impl == null) {
            throw new ServletException("Missing impl attribute on
handler element.");
        }
        NodeList params =
handlerElem.getElementsByTagName("param");
        HashMap paramMap = new HashMap();
        for (int j=0;j<params.getLength();j++) {
            Element paramElem = (Element) params.item(j);
            String name = paramElem.getAttribute("name");
            String value = paramElem.getAttribute("value");
            paramMap.put(name, value);
        }
        // Create a class and stick it in the hashtable
        Class handlerClass = Class.forName(impl);
        RestHandler handler = (RestHandler)
handlerClass.newInstance();
        handler.init(paramMap);
        handlers.put(path, handler);
        System.out.println("Created handler for path "+path);
    }
} catch (SAXException ex) {
    throw new ServletException(ex);
} catch (IOException ex) {
    throw new ServletException(ex);
} catch (ClassNotFoundException ex) {
    throw new ServletException(ex);
} catch (IllegalAccessException ex) {
    throw new ServletException(ex);
} catch (InstantiationException ex) {
    throw new ServletException(ex);
} catch (RestException ex) {
    throw new ServletException(ex);
}
}

public void destroy() {
    Enumeration e = handlers.elements();
    while (e.hasMoreElements()) {
        RestHandler h = (RestHandler) e.nextElement();
    }
}

```



```

        try {
            h.destroy();
        } catch (RestException ex) {
            ex.printStackTrace();
        }
    }
}

```

13. The `init()` method loads a meta file that is located in the `WEB-INF` directory. The meta file will contain the definitions for accessing the REST resource.

14. Create a file called `rest-handlers.xml` in the `WebContent/WEB-INF` directory and add the snippet of configuration information shown in Example 8-5.

*Example 8-5 rest-handlers.xml configuration file for REST servlet*

```

<?xml version="1.0" encoding="UTF-8"?>
<rest-handlers>
    <handler impl="com.ibm.itso.rest.handlers.demo.DemoRestHandler"
path="/employee/demo">
        </handler>
</rest-handlers>

```

15. Now that the servlet can load the configuration the next step is to implement the method that will handle the requests as they arrive for the defined URL mapping of `/rest/*` (that is, any URIs that are prefixed by this URI). To do this add the code snippet shown in Example 8-6.

*Example 8-6 handleRequest method for RestProviderServlet*

```

private void handleRequest(int operation, HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    HashMap reqParms = new HashMap();
    Enumeration e = request.getParameterNames();
    while (e.hasMoreElements()) {
        Object key = e.nextElement();
        reqParms.put(key, request.getParameter((String) key));
    }
    String path = parsePath(request);
    System.out.println("Handling request for path "+path);
    RestHandler handler = (RestHandler) handlers.get(path);
    if (handler == null) {
        throw new ServletException("Missing handler for path "+path);
    }
    try {
        handler.handleOperation(operation, path, reqParms,
            request.getInputStream(), response.getOutputStream());
    }
}

```

```

        System.out.println("Handled request for path "+path+" OK");
    } catch (RestException ex) {
        throw new ServletException(ex);
    }
}

```

---

16. With the core functionality now implemented, overriding the core servlet interface is required to map the REST operations to the specific RestHandler operations.

17. Add the code shown in Example 8-7 to the servlet.

*Example 8-7 REST operations for RestProviderServlet*

---

```

protected void doDelete(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    handleRequest(RestHandler.OPERATION_DELETE, request, response);
}

protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    handleRequest(RestHandler.OPERATION_READ, request, response);
}

protected void doPost(HttpServletRequest request,
HttpServletResponse response) throws ServletException, IOException {
    String action = request.getParameter("action");
    if (action != null) {
        if (action.equalsIgnoreCase("create")) {
            handleRequest(RestHandler.OPERATION_CREATE, request,
response);
        } else if (action.equalsIgnoreCase("delete")) {
            handleRequest(RestHandler.OPERATION_DELETE, request,
response);
        }
    }
    handleRequest(RestHandler.OPERATION_UPDATE, request, response);
}

protected void doPut(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
    handleRequest(RestHandler.OPERATION_CREATE, request, response);
}

```

---

18. The final step is to add the method that will parse the URI path. The reason for adding this into a separate method is so that it can later be extended. Add the `parsePath()` code shown in Example 8-8 to the `RestProviderServlet`.

*Example 8-8 parsePath() method*

---

```
/**
 * Returns the REST path portion of the URL requesting the servlet,
 * e.g. converts
 *
<code>http://localhost:8080/RestProvider/rest/employee/salary</code>
into <code>/employee/salary</code>.
 */
private String parsePath(HttpServletRequest request) {
    return request.getPathInfo();
}
```

---

This completes the creation of the initial phase of the REST servlet.

### **Adding a demo REST Handler**

To help with testing, follow these steps to implement a sample REST Handler:

1. The REST Handler uses the JSON4J libraries. Copy the `JSON4J.jar` into the `WEB-INF/lib` directory.
2. Add a new class named `DemoRestHandler` to the `com.ibm.itso.rest.handlers.demo` package.
3. The REST Handler sends a JSON format string back to the requester. To accomplish this add the code snippet shown in Example 8-9 to the `DemoRestHandler` class.

*Example 8-9 DemoRestHandler code snippet*

---

```
package com.ibm.itso.rest.handlers.demo;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.HashMap;

import com.ibm.itso.rest.RestEntity;
import com.ibm.itso.rest.RestException;
import com.ibm.itso.rest.RestHandler;
import com.ibm.json.java.JSONArray;
import com.ibm.json.java.JSONObject;
```

```

public class DemoRestHandler implements RestHandler {

    public void destroy() throws RestException {
        System.out.println("destroy-ed OK");
    }

    public void handleOperation(int operation, RestEntity restEnt,String
path, HashMap parameters,
        InputStream inStream, OutputStream outStream) throws
RestException {

        switch (operation) {
            case RestHandler.OPERATION_READ: {

                JSONArray array = createSampleData();

                try {

                    String jsonStr = array.serialize(true);
                    outStream.write(jsonStr.getBytes());

                } catch (IOException ex) {
                    throw new RestException(ex);
                }
                break;
            }
        }
    }

    public void init(HashMap params) throws RestException {
        System.out.println("init-ed OK");
    }

    public JSONArray createSampleData()
    {
        JSONArray array = new JSONArray();
        JSONObject obj = new JSONObject();
        obj.put("firstname", "Matthew");
        obj.put("surname","Perrins");
        obj.put("title", "Executive IT Specialist");
        obj.put("email", "matthew_perrins@uk.ibm.com");
        array.add(obj);

        obj = new JSONObject();
    }
}

```

```

        obj.put("firstname", "Robin");
        obj.put("surname", "Howlett");
        obj.put("title", "Software Engineer");
        obj.put("email", "HOWLETTTR@ie.ibm.com");
        array.add(obj);

        return array;
    }
}

```

---

The `DemoRestHandler` contains a method named `handleOperation` that is called whenever a REST resource URI is found to match the REST Handler. The `DemoRestHandler` is passed a set of parameters, including the REST operation, which is used to determine how to process the PUT, GET, POST, or DELETE request.

4. To test the REST resources, create a simple HTML file named `index.html` in the root of the `WebContent` directory and add the HTML shown in Example 8-10.

*Example 8-10 The `index.html` file in the Web project root*

---

```

<html>
<body>
<h3>Samples</h3>
<ol>
<li><a href="rest/employee/demo">Demo</a>
</body>
</html>

```

---

5. Save the `index.html` file and then run this Web project and test the Demo link.
6. If successful, you should see a Web page containing JSON text similar to the one shown in Figure 8-3.

```

[ { "email": "matthew_perrins@uk.ibm.com", "firstname": "Matthew", "surname":
"Perrins", "title": "Executive IT Specialist" }, { "email": "HOWLETTTR@ie.ibm.com",
"firstname": "Robin", "surname": "Howlett", "title": "Software Engineer" } ]

```

*Figure 8-3 Demo REST Handler output*

You now have the basic building blocks for creating a fully functional REST Handler servlet for your J2EE applications. The following sections focus on adding Entity support and the ability to invoke Groovy scripts. These scripts will

replace the REST Handler interface and allow simple Groovy scripts to perform very similar tasks.

## Adding entity support

Using resource URLs and adding entity elements to them is a very simple yet powerful way of using the fabric of the Internet to consume RESTful services. The following section covers how the RESTful servlet can be extended to support entities.

The four most common entity types are:

- ▶ Retrieve an entity by using a key.
- ▶ Retrieve an entity by using an index.
- ▶ Retrieve an entity by using a range.
- ▶ Retrieve an entity by using a list of index numbers.

This set of entity types enables a wide range of client-side actions to be mapped around these resource entity types.

To support entities you need to parse the URI resource reference and then use regular expressions to decide what type of entity request it is. Once a match is found you need to parse the URI and retrieve the data needed to inform the REST Handler of how it should process the entity request.

To enable entity support in the example RESTful servlet:

1. Create an `RestEntity` class that will be used to hold the meta data for the entity request. Add the class to the `com.ibm.itso.rest` package and use the code snippet shown in Example 8-11.

### *Example 8-11 RestEntity class*

---

```
package com.ibm.itso.rest;

public class RestEntity
{
    public static final int ENTITY_KEY = 0;
    public static final int ENTITY_INDEX = 1;
    public static final int ENTITY_RANGE = 2;
    public static final int ENTITY_LIST = 3;

    private int type = -1;
    private String keyvalue = "";
    private int start = -1;
    private int end = -1;
```

```

    private int index = -1;
    private int[] list = null;
}

```

- Using Rational Application Developer tooling select **Context Menu** → **Source** → **Generate Getters and Setters** and select all the private field members.
- When the code generation has completed save the class.
- Modify the `RestHandler.handleOperation` interface to handle the passing of the `RestEntity` class to the REST Handler. Open the `RestHandler.java` class.
- Replace the `handleOperation` method with the code snippet shown in Example 8-12.

*Example 8-12 New handleOperation method on RestHandler*

```

/**
 * Handles the access to a resource based on the provided I/O
 * streams and parameters.
 */
public void handleOperation(int operation, RestEntity restEntity,
    String path, HashMap parameters, InputStream inStream, OutputStream
    outStream) throws RestException;

```

- The next step is to modify the `RestProviderServlet` so that it can handle the new Entity requests as they arrive with the URI resource reference. The simplest means for determining the type of entity request is to use a regular expression to compare against the URI to determine whether it matches the entity style. We used the four regular expressions listed in Table 8-4.

*Table 8-4 Entity types and the matching using a regular expression*

Type	URI entity resource	Regular expression
KEY	/employees/perrins	/[a-zA-Z]*
INDEX	/employees/1	/[0-9]*
RANGE	/employees/1..3	/[0-9]*[.][.][0-9]*
LIST	/employees/	/([0-9]*[.])+[0-9]*

- Open the `RestProviderServlet.java` class in the `com.ibm.itso.rest.servlet` package.

8. Add the array of regular expressions shown in Example 8-13 to the beginning of the class.

*Example 8-13 Regular expressions for entity handling*

---

```
Pattern[] patterns = new Pattern[] {  
    Pattern.compile("/[a-zA-Z]*"),  
    Pattern.compile("/[0-9]*"),  
    Pattern.compile("/[0-9]*[.][.][0-9]*"),  
    Pattern.compile("/([0-9]*[,])+[0-9]*") };
```

---

9. To enable the support for entities modify the `handleRequest` method so that it can receive the parsed URI reference and then check which entity it supports. To do this, replace the `handleRequest` method with code that will perform the URI entity checking. Use the code snippet shown in Example 8-14 to replace the method in the `RestProviderServlet` class.

*Example 8-14 Code snippet*

---

```
private void handleRequest(int operation, HttpServletRequest  
request,  
    HttpServletResponse response) throws ServletException,  
IOException {  
    HashMap reqParms = new HashMap();  
    reqParms.put("_servletContext", context);  
    Enumeration e = request.getParameterNames();  
    while (e.hasMoreElements()) {  
        Object key = e.nextElement();  
        reqParms.put(key, request.getParameter((String) key));  
    }  
    String path = parsePath(request);  
    System.out.println("Handling request for path " + path);  
    RestHandler handler = null;  
    RestEntity restEnt = null;  
    if (handlers.get(path)!=null)  
    {  
        handler = (RestHandler)handlers.get(path);  
    } else {  
        String entity = path.substring(path.lastIndexOf("/"));  
        restEnt = checkEntity(entity);  
        path = path.substring(0,path.lastIndexOf("/"));  
        handler = (RestHandler)handlers.get(path);  
    }  
  
    if (handler == null) {
```



```

        throw new ServletException("Missing handler for path " +
path);
    }
    try {
        handler.handleOperation(operation, restEnt, path, reqParms,
request
        .getInputStream(), response.getOutputStream());
        System.out.println("Handled request for path " + path + "
OK");
    } catch (RestException ex) {
        throw new ServletException(ex);
    }
}

```

---

10. The next step adds the checking of the URI to determine whether it maps directly to a REST Handler or whether it contains entity elements that need to be processed. To check the entity elements, a new method, `checkEntity`, must be added to the `RestProvider` servlet class to validate the URI. Add the code shown in Example 8-15 at the bottom of the servlet.

*Example 8-15 checkEntity method for parsing entity elements on a REST URI*

---

```

private RestEntity checkEntity(String path) {
    RestEntity restEnt = new RestEntity();
    int count=0; boolean foundpattern=false;
    while(!foundpattern && count < patterns.length){
        Pattern pattern = patterns[count];
        Matcher match = pattern.matcher(path);
        boolean found = match.matches();
        if (found) {
            restEnt.setType(count);
            switch (count) {
                case RestEntity.ENTITY_KEY: {
                    restEnt.setKeyvalue(path.substring(1));
                    foundpattern = true;
                    break;
                }
                case RestEntity.ENTITY_INDEX: {
                    int index = -1;
                    try
                    {
                        index = Integer.parseInt(path.substring(1))-1;
                        restEnt.setIndex(index);
                    }
                    catch (NumberFormatException ex) {
                        // ignore
                    }
                }
            }
        }
    }
    return restEnt;
}

```

```

        } catch (Exception e) {
            System.out.println("Problem with Entity index");
        }
        foundpattern = true;
        break;
    }
    case RestEntity.ENTITY_RANGE: {

        int start = -1;
        int end    = -1;
        try
        {
            int dot = path.indexOf(".");
            start = Integer.parseInt(path.substring(1,dot))-1;
            end = Integer.parseInt(path.substring(dot+2))-1;
            restEnt.setStart(start);
            restEnt.setEnd(end);

        } catch (Exception e) {
            System.out.println("Problem with Entity Range");
        }
        foundpattern = true;
        break;
    }
    case RestEntity.ENTITY_LIST: {

        StringTokenizer tok = new
StringTokenizer(path.substring(1),",");
        int[] list = new int[tok.countTokens()];
        int i = 0;
        while(tok.hasMoreTokens())
        {
            String token = (String)tok.nextToken();
            try
            {
                list[i++] = Integer.parseInt(token)-1;
            } catch (Exception e)
            {
                System.out.println("Problem with Entity list");
            }
        }
        restEnt.setList(list);
    }
    foundpattern = true;
    break;
}

```

```

        }
    }
    count++;
}
if (!foundpattern)
    restEnt = null;

return restEnt;
}

```

---

Let us examine the function of this method. The method loops through the patterns that have been precompiled and stored in the `patterns` instance variable. The order of the patterns matches the numbering sequence given to the entity type's static variables in the `RestEntity` class. The entity URI is matched against each of the regular expressions. If a match is found the instance of the `RestEntity` class is populated with the values required to process the specific entity. This class is then returned. If the class is null then it is assumed that the REST URI is a direct match to the definition in the `rest-handler.xml` file.

With the `RestProviderServlet` now updated you need to implement the entity support within the REST handler.

11. Open the file **WebContent/WEB-INF/rest-handlers.xml**.

12. Add the configuration statements to the file as shown in Example 8-16.

*Example 8-16 Configuration for EmployeeRestHandler*

---

```

<handler impl="com.ibm.itso.rest.handlers.demo.EmployeeRestHandler"
path="/employees">
</handler>

```

---

13. Save the `rest-handlers.xml` configuration file.

The `EmployeeRestHandler` must now be implemented. You can decide with how much detail you want it to be implemented. With the total number of REST operations equalling four and with the total number of entity types equalling four then this class may grow large in size. You can decide how you want to implement this in the future. For demonstration purposes we include the GET operation and the four entity operations all in one class.

14. Create a new class called `EmployeeRestHandler` in the `com.ibm.itso.rest.handlers.demo` package and add the code shown in Example 8-17.

*Example 8-17 EmployeeRestHandler implementation code*

---

```
package com.ibm.itso.rest.handlers.demo;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.HashMap;
import java.util.Iterator;

import com.ibm.itso.rest.RestEntity;
import com.ibm.itso.rest.RestException;
import com.ibm.itso.rest.RestHandler;
import com.ibm.json.java.JSONArray;
import com.ibm.json.java.JSONObject;

public class EmployeeRestHandler implements RestHandler {

    public void destroy() throws RestException {
        System.out.println("destroy-ed OK");
    }

    public void handleOperation(int operation, RestEntity restEnt, String
path, HashMap parameters,
        InputStream inStream, OutputStream outStream) throws
RestException {
        switch (operation) {
            case RestHandler.OPERATION_READ: {

                try {

                    String jsonStr = processEntity(restEnt);
                    outStream.write(jsonStr.getBytes());

                } catch (IOException ex) {
                    throw new RestException(ex);
                }
                break;
            }
        }
    }
}
```

```

    public void init(HashMap params) throws RestException {
        System.out.println("init-ed OK");
    }
}

```

---

The OPERATION\_READ operation is invoked when a standard GET operation is received from a Web browser. The code will invoke the processEntity method by passing an instance of the RestEntity object that is passed into the handler.

15. To complete the EmployeeRestHandler class add the code for the processEntity method as shown in Example 8-18.

*Example 8-18 processEntity method*

---

```

public String processEntity(RestEntity restEnt) throws IOException
{
    JSONArray array = createSampleData();
    String resultJSON = "";

    if (restEnt == null) {
        resultJSON = array.serialize();
    } else {

        switch (restEnt.getType()){

            case RestEntity.ENTITY_KEY:{

                Iterator i = array.iterator();
                while(i.hasNext())
                {
                    JSONObject obj = (JSONObject)i.next();
                    String key = (String)obj.get("surname");
                    if (key.toLowerCase().equals(restEnt.getKeyvalue()))
                    {
                        resultJSON = obj.serialize();
                    }
                }
                break;
            }
            case RestEntity.ENTITY_INDEX:{
                resultJSON =
                ((JSONObject)array.get(restEnt.getIndex())).serialize();
                break;
            }
        }
    }
}

```

```

    }

    case RestEntity.ENTITY_RANGE:{
        JSONArray listArray = new JSONArray();
        for(int r=restEnt.getStart();r<=restEnt.getEnd();r++)
            listArray.add(((JSONObject)array.get(r)));
        resultJSON = listArray.serialize();
        break;
    }
    case RestEntity.ENTITY_LIST:{
        JSONArray listArray = new JSONArray();
        for(int r=0;r<restEnt.getList().length;r++)
            listArray.add(((JSONObject)array.get(restEnt.getList()[r])));
        resultJSON = listArray.serialize();
        break;
    }
}
}
return resultJSON;
}

```

---

The processEntity method will support each of the four entity operations performed on the REST URI:

- ▶ If an ENTITY\_KEY operation, the method compares the RestEntity.getKeyvalue() value against the surname property in the JSON test data. If a match occurs it then serializes the JSONObject.
- ▶ If an ENTITY\_INDEX operation, the method checks the RestEntity.getIndex() value and uses it as an index into the JSONArray objects from the test data.
- ▶ If an ENTITY\_RANGE operation, the method navigates from the RestEntity.getStart() to the RestEntity.getEnd() index elements within the JSONArray objects.
- ▶ If an ENTITY\_LIST operation, the method extracts the elements using the index data stored in the RestEntity.getList(). It creates a new JSONArray and stores the objects before serializing them back in the response.

16. The final step is to update the HTML file used to test the REST URIs. Open the file `index.html` in the `WebContent` root directory and replace the contents with the code shown in Example 8-19.

*Example 8-19 The `index.html` sample file*

---

```
<html>
<body>
<h3>Samples</h3>
<ol>
<li><a href="rest/employees">Employees</a>
<li><a href="rest/employees/akin">Employee with key of akin</a>
<li><a href="rest/employees/howlett">Employee with key of howlett</a>

<li><a href="rest/employees/1">First Employee</a>
<li><a href="rest/employees/1..3">First Three Employees</a>
<li><a href="rest/employees/1,3">First and Third Employees</a>
</body>
</html>
```

---

Notice that each of the use cases for the REST entity support in the REST servlet can be tested.

You are now in a position to test the various entity-based REST URIs that you have created.

## Summary

You have completed the creation of one type of entity-based REST servlet. This enables you to define REST URIs and map them to Java-based REST Handlers. You have also added specific REST entity support to implement a mechanism for accessing a REST URI using common entity operations that allows you to access data by using a key value, an index, a range, or a list of index elements. This REST servlet should give you an idea of the complexity of creating a service servlet that supports your application's needs. It may not be long before these areas become standardized and the programming models support these types of operations natively. If you look at the WebSphere sMash programming model (Project Zero: <http://www.projectzero.org>) you will see that one of its key advantages is that its programming model is defined around entity REST principles, which makes it possible to create Groovy scripts that directly link to the REST HTTP operations. This provides a quick method for application and middleware service development.

## 8.3 REST-as-CRUD or REST-as-protocol

Although the practice of demonstrating REST by applying CRUD principles (called resource-oriented architecture, or ROA) is often used to ease an unfamiliar reader into the concepts of REST, it is not the only way, and indeed perhaps not even the recommended way of working with REST. ROA may work best in a data-centric approach but does not always lend itself to conceptualizing business processes and resource interaction.

Today, the notion of what the Web represents, or can represent, is changing. The Web has evolved from a document-publishing system to a more obtuse merging of a publishing system and an application platform, which has been cajoled into being structured similarly to the old system. REST shifts it to another level—a new type of Web entity. Certainly, REST can still be used for document publish and retrieval, as well as serve a collection of document-like resources, but once the focus switches away from the traditional Web conceptual model to application concepts that are *of* the Web, powerful opportunities arise.

An excellent description of the power of this mind shift comes from Tim Ewald<sup>1</sup>:

Every communication protocol has a state machine. For some protocols they are very simple, for others they are more complex. When you implement a protocol via RPC, you build methods that modify the state of the communication. That state is maintained as a black box at the endpoint. Because the protocol state is hidden, it is easy to get things wrong...

The essence of REST is to make the states of the protocol explicit and addressable by URIs. The current state of the protocol state machine is represented by the URI you just operated on and the state representation you retrieved. You change state by operating on the URI of the state you are moving to, making that your new state. A state's representation includes the links (arcs in the graph) to the other states that you can move to from the current state. This is exactly how browser based apps work, and there is no reason that your [application's] protocol cannot work that way too.

What this means is that due to the nature of REST, in terms of a REST-as-protocol approach, each unique state in the state machine is uniquely identified and addressed via the URI. Therefore, the endpoints of the application may be distributed, constructed, and organized in whatever manner you deem fit. The sheer scalability, control, and durability, through a clear and familiar format, offered by such an approach cannot be understated. Replicating such an approach using RPC would be far more complex.

---

<sup>1</sup> <http://www.pluralsight.com/blogs/tewald/archive/2007/04/26/46984.aspx>



Using URIs as state transition agents allows for a uniform method of state traversal while also permitting the ability to embed transition URIs within state representations. RPC is comfortable, as it follows the traditional approach to method invocation. However, even though HTTP follows a request/response model, you are not constricted by the call-stack method of interaction. A stack frame does not need to be copied from endpoint to client. The messages are discrete in themselves and use the nature of URIs for state transitions. This is not to say that REST has usurped RPC, but there are significant benefits to REST that are difficult to replicate with RPC. However, as the protocol state becomes more complex and the process time frame increases, it becomes essential to manage that state, and REST can do this more simply.

As previously stated, REST-as-CRUD does have a place, especially with data-centric applications and indeed perhaps the majority of current RESTful applications are designed as read-only data providers. The two approaches may even exist in tandem in the same application over a variety of interfaces. ROA will mean that transactions, as we now know them, are not provided for and, as such, for many, it will not be applicable for the business interface. However, transactions may not be needed. The Atom Publishing Protocol (APP) is a highly usable system for entity REST and does not need transactions. They can be provided for by utilizing a series of GET and POST/PUT requests. The fact that this is a grouping of requests means that it now enters the REST-as-protocol domain.

### **REST is not XML-RPC**

It is difficult to introduce the concepts of REST and provide a sample scenario that could also be implemented using other approaches, like XML-RPC. Indeed, many of the REST samples basically appear the same as XML-RPC except for the embedding of URIs in state representations. However, REST fundamentally changes the concept of how states are transitioned and when loose-coupling, scalability, and reliability are paramount, REST offers a powerful architecture.

## **8.4 REST or SOA: Making the decision**

It is beyond the scope of this book to compare the merits of REST and SOA. We have already outlined how REST can contribute to potential improvements in scalability and interoperability. Some additional advantages of REST that can be identified in common practice today are presented.

Many SOA implementations are fully XML-based solutions, meaning that significant parsing is required versus the REST's greater focus on more efficient generation of representations (for example, URIs). Additionally, the use of the

URI for representing states has value also in the ability to detail branches of the resource pool tree.

Interoperability improvements are dependent on defining standards, and in an enterprise environment we have seen attempts for such agreements fail to materialize. Also, some business models are complex entities and, while there may be some entity overlap with the general industry, the core entity is unique to the business. So the question must be asked as to whether REST really supersedes a particular incumbent architecture. A counter-argument is that the growth of Web 2.0 has shown that perhaps in the future consumers will demand interoperability and standards. But that remains to be seen.

Complex business logic may be composed of a subset of general entities. However, extending base entities efficiently may be difficult architecturally. A mechanism for sharing entities (that is, a common entity library) effectively has proven to be a significant design issue, one that is related to the problems of moving towards a more semantic Web.

There are no set rules for building or converting existing applications to RESTful applications but there are a number of heuristic approaches that can be used. For instance, a simple scenario of replacing an SOA implementation with multitudes of get\* methods utilizing the HTTP GET and the URI solution may appear to be essentially the same, apart from the previously mentioned embedded URIs. URIs also lend themselves to improved distribution among users.

The ability to use content-type is dependent on the presumption of common entities and again would benefit from standardization. URIs are a natural fit for architecting for scale, versioning, and load-balancing, as they can be partitioned easily for resource management. Partitioning can also be done through the data, but it tends to be quite application specific.

When REST uses HTTP (almost all the time, but in truth, it does not have to be) you can take advantage of its existing benefits including streaming, non-XML mime types, and caching abilities, again benefitting scalability. In a Web 2.0 world, however, effectively caching dynamic data is extremely challenging and indeed may be deemed extraneous versus correctness, especially regarding private personal data. Also, it might be argued that most of the caching performed is in fact application specific and its value is highly dependent then on the application logic. Caching presents significant development challenges and REST may not necessarily do anything to ease those challenges.

REST can reduce the complexity of conceptualizing data interaction. When the meaning of the data sought is known (a noun), then how to change it (with a verb), if allowed, is known. Therefore, aspects of entities lead themselves to being derived. Data constraints in REST need also to be considered. Consider

the Atom Publishing Protocol, as it has a defined syntax and discrete set of semantics. Yet the author entity field is not defined as anything other than an identifier (for example, e-mail address, user name). If the identity scheme is proprietary and non-public, as most tend to be, then this protocol cannot define acceptable attribute values, preventing entity reuse.

REST does support authentication but credential management is deemed a more important issue for enterprises than message format. Amazon's S3 REST API uses the standard HTTP Authorization header to pass authentication information, and a custom HTTP scheme based on a keyed-HMAC (Hash Message Authentication Code) for authentication. As a side note, Amazon's Web services have both REST and SOAP interfaces, but 85% of their services are called using the REST interface.

### **POST versus PUT/DELETE**

In practice, supporting the GET and POST functions is generally enough to be considered REST compliant. Not all network architectures support PUT/DELETE (a consequence of POST getting preference in HTML forms rather than any optimization choice) and many developers take the view that "GET=read, POST=write". However, that is not to say that PUT/DELETE does not have a place in a RESTful application. But developers have tended to focus more on the quality of the action being undertaken (and correct design of process) than the verb being used. AtomPub uses POSTing to a collection with the server when the request is required to submit the ID and it is the server that checks whether that ID is valid (a style of conditional-POST), as opposed to PUTting to a specific resource URI.

## **8.4.1 Reliable messaging**

Both PUT and DELETE, unlike POST, are naturally idempotent, meaning that sending those requests more than once to the same resource will not change their state. In other words, it can handle the request arriving multiple times. Since the network should always be assumed to be unreliable, it is good practise to design each step of a process to use techniques that are either idempotent or transactional, so that the concept of a message being dealt with only once is maintained. This technique may be further assisted by logging the IDs of the messages being sent. In fact, if it is logged to a durable local store as part of a local transaction then you have a reliable messaging technique.

Once a RESTful implementation is selected, a decision about the granularity of resources should be considered. For instance, using micro resources (where a URI represents the significant fields of items) is one approach. Even though the type space is significantly expanded, it can be argued that the internal structure has been made externally accessible.

## 8.4.2 Transactional REST

Building a RESTful implementation does not imply having to discard the fundamental concepts of transactionality (that is, atomicity, consistency, isolation, and durability (ACID)). Transactions can also be thought of as states, with the actions that constitute a transaction representing the transitioning of the state to a defined representation.

Consider the following actions:

- ▶ GETting a resource
- ▶ POSTing some changes to that resource (affecting other resources)
- ▶ GETting other resources to check effects
- ▶ Committing the transaction

The transaction concept can be mapped to REST by also treating the transaction as a resource. For instance:

- ▶ GET `http://itso.ibm.com/tx`

This returns a URI of a newly created transaction resource (for example, `http://itso.ibm.com/tx/393423`)

- ▶ GET `http://itso.ibm.com/tx/393423`

This returns an XML representation of transaction, for example:

```
<transaction>
  <state>pending</state>
  <txlog />
</transaction>
```

- ▶ Anytime a resource is interacted with as part of this transaction the context used contains the transaction URI `http://itso.ibm.com/tx/393423/res1`. This view of the resource is only available to the transaction owner and all others would see the default resource outside of the tx context.
- ▶ Each time an action is undertaken, it can be logged to the transaction log:

```
<transaction>
  <state>pending</state>
  <txlog>
    <action id="1" type="POST"
uri="http://itso.ibm.com/tx/393423/res1" />
    <action id="2" type="POST"
uri="http://itso.ibm.com/tx/393423/res2/something" />
  </txlog>
</transaction>
```

- ▶ A particular action can therefore be rolled-back to a desired state by modifying the transaction resource to remove the actions that now need to be rolled back.
- ▶ Transactions can be marked successful or failed (depending on the success of the actions undertaken within the transaction) by GETting the transaction resource, setting the state to success or failed, and POSTing back the modified transaction. The HTTP response can then be used to identify what to do next (whether the transaction completed successfully, if certain steps had a problem and were identified, or that nothing was changed and the transaction failed).
- ▶ Like other resources, transactions can be deleted, modified, and created using GET, POST, PUT, and DELETE (if implemented).

## 8.5 REST and Ajax

Ajax has enjoyed tremendous popularity in recent years. However, many organizations that attempt to incorporate Ajax into their applications, by either retro-fitting existing frameworks or developing new Ajax-enabled frameworks, have experienced serious difficulties with their architecture and performance. We have outlined how the REST architecture can assist with application design and now we detail the considerations applicable for harmonizing REST and Ajax.

Ajax technology has massively increased the immersiveness of Web applications, but its approaches often violate the principle concepts of REST by reducing scalability and introducing greater complexity.

REST has a *client-stateless-server* constraint meaning that the user session state is forbidden to exist on the server. This constraint is intended to promote the properties of visibility, reliability, and scalability. On the flip-side, immersive Web applications are fundamentally focused on providing high levels of personalization to a single user. Many developers therefore believe a choice must be made:

- ▶ Comply with the constraint and send all state data with each request.
- ▶ Store session data on the server, let the client send a session identifier, and supply the known session data that matches.

The former is seen as too much of a development challenge. The latter design is what is favored but places significant strain on application architecture. A good example is Java Servlet's HttpSession API, which lets you associate session state with particular users. However, as this associated data increases, memory resources requirements also grow, forcing architectural changes like clustering,

which then requires serialization to be considered and finally the need to persist session data in case of architectural fall-overs.

REST also has significant strength at resource caching due to a well-defined resource node graph. Since Ajax applications are so dynamic, a server cannot reasonably hold state for each combination of application. Caching is therefore sacrificed, increasing the strain on the system.

One solution to this problem is to cache finely grained resources so that an Ajax document is a coarsely grained collection of pre-assembled parts. However, that means that every request results in non-trivial server processing. Therefore, to prevent non-essential server requests (for example, search bots) since they are resource expensive, REST can simply return a Not-Modified message after the initial server request.

Before Ajax, distributing processing and memory requirements to the client was not feasible, as application state was destroyed at each new page request. Ajax has not enabled the potential for stateful clients. However, by locating presentation and data binding on the client, a completely different architectural style can be used, one that is entirely conducive to REST and also more robust, as there is less dependency on the server.

A well-designed Ajax engine, one that contains a great deal of application logic and presentation framework elements without containing business data or personalized content, can be highly cachable. The Dojo Toolkit, contained within the Ajax client runtime (see Chapter 9, “Ajax client runtime” on page 269) packaged with the Feature Pack for Web 2.0, is a good example. In Dojo, build time tools create a compressed JavaScript file with all the application’s logic, presentation, and styling as a single file. This means that not only is it cachable (meaning re-visits are much quicker), it is also easy to configure for use with proxies.

In terms of how an application would appear when REST and Ajax have been harmonized depends on the application type. Consider the following scenario: A user searching for a flight could have separate requests for general data and user-specific data:

```
GET http://itso.ibm.com/flights/flight?dep=DUB&arr=NC
GET http://itso.ibm.com/users/robin/flights/flight?dep=DUB&NYC
```

The first would contain general flight data that is not user specific and is therefore easily cached. The second request returns specific personalized data (such as links to hotel resources or rental car resources you have used before, or recommend one from other users’ experiences). The second request would only be accessible by me (some authorization would be required), but the point is that

by having a URI for my own personalized data it does not interfere with the application's general caching ability.

## 8.6 REST in the industry

In this section we discuss REST in the industry.

### 8.6.1 JAX-RS (JSR 311)

JAX-RS<sup>2</sup> is being developed as an API for providing support for RESTful Web services in the Java platform and is currently in the 1.0 Early Draft Review phase. The JSR aims to provide a high-level API for easy-to-write RESTful Web services that are not dependent on the underlying technology. The API supports using annotations for a declarative programming style.

### 8.6.2 Jersey

Jersey is an open source (CDDL licence) implementation of the JAX-RS API. You use annotations to map resources and methods to URIs (that is, `@UriTemplate("/users/robinhowlett")`) and to indicate responses' content types (that is, `@ProduceMime("application/json")`). EntityProviders are responsible for reading and writing requests and responses as objects to handle specific types such as JSON objects, XML, and Atom. You do not need to extend particular classes or implement any interfaces, making it extremely conducive to unit testing.

### 8.6.3 Restlet

Restlet is a lightweight REST framework for Java. It is an open source (albeit requiring a Joint Copyright Assignment) project distributed with a CDDL license or an LGPL license. It consists of a Restlet API, a reference implementation, and a set of extensions.

Restlet was designed as an alternative to using the servlet API for RESTful implementations. The servlet API requires mapping URI patterns and their respective handlers in a central configuration file. Restlet was also designed to provide additional control over I/O optimization (by supporting NIO) and HTTP caching and content negotiation. Each concept of REST (as outlined by Fielding) has a corresponding Java class in the Restlet API. The framework also supports JAX-RS.

---

<sup>2</sup> <http://jcp.org/aboutJava/communityprocess/edr/jsr311/index.html>





## Ajax client runtime

This chapter describes how the Ajax client runtime of the WAS Feature Pack for Web 2.0 is used to facilitate client-side programming for Web applications. It includes details on the following topics:

- ▶ Introduction to the IBM Dojo Toolkit
- ▶ Overview of the IBM Dojo Toolkit architecture
- ▶ Setting up Dojo in a Web application
- ▶ Overview and usage of Dojo core libraries
- ▶ Overview and usage of Dojo widget libraries
- ▶ Overview and usage of Dojo's extension libraries
- ▶ Instructions on how to build customized widgets
- ▶ Asynchronous client/server communication with Dojo's XHR-functions
- ▶ Linking widgets together with the dojox.wire package
- ▶ Publishing and subscribing to JavaScript events with Dojo
- ▶ Data access with the dojo.data package
- ▶ JavaScript unit testing using the Dojo Objective Harness
- ▶ IBM widget extensions to the Dojo Toolkit
- ▶ JavaScript file compression with ShrinkSafe
- ▶ Plants for WebSphere sample

## 9.1 Introduction to the Dojo Toolkit

This section provides an overview of the Ajax client runtime together with the Dojo Toolkit.

### 9.1.1 Overview of the Ajax client runtime

The Ajax client runtime is a best-in-class Ajax development toolkit for the WebSphere Application Server. It is based on an IBM supported version of the Dojo Toolkit.

**Note:** The version of the Dojo Toolkit provided by the Ajax client runtime described in this chapter is 1.0. Dojo 1.1 was released while writing this book. Any changes in the latest version that affect the topics being covered in the following sections are highlighted. All changes from 1.0 to 1.1 are detailed at:

<http://dojotoolkit.org/book/dojo-1-1-release-notes>

Dojo 1.1 is backwards compatible with 1.0. You can also run all provided examples with Dojo 1.1.

In addition to the base functionality of the Dojo Toolkit, the Ajax client runtime comes with additional IBM enhancements to simplify working with different protocols, data formats, and connectivity options. The runtime includes extensions to the Dojo Toolkit, which includes a SOAP library, Gauge widgets, the Atom library, and the OpenSearch library.

Figure 9-1 shows an overview of the Ajax client runtime.

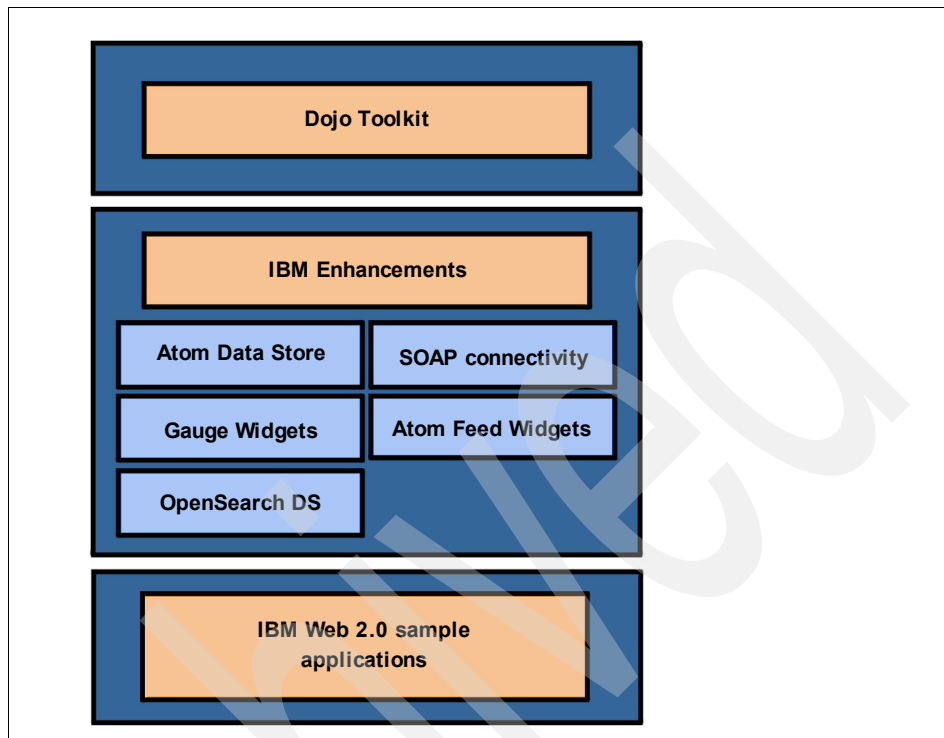


Figure 9-1 Overview of the Ajax client runtime

### 9.1.2 Overview of the Dojo Toolkit

The Dojo Toolkit is a multi-platform, open source JavaScript toolkit backed by IBM, Sun Microsystems, SitePen, AOL, and other companies. It is not tied to any specific server-side toolkit. Therefore, it can be used in conjunction with PHP, Java servlets, Java Server Pages, ASP, and so on, or even with mobile devices.

The Dojo Web site is available at:

<http://www.dojotoolkit.org/>

You can check that Web site for the latest information regarding the Dojo Toolkit.

This chapter describes the main concepts of Dojo, and therefore does not provide an API description of the toolkit. The Dojo Toolkit APIs can be found at:

<http://api.dojotoolkit.org>

The Dojo toolkit has a modular design:

- ▶ Base: The kernel of the toolkit. Dojo Base bootstraps the toolkit. It includes:
  - A module loader
  - Language utilities, for example, array functions, functions for declaring classes, and inheritance
  - Query, node, and CSS utilities
  - Functions for cross-browser communication, for example, XHR
  - JSON serialization/deserialization
  - A cross-browser event and a publish/subscribe system
  - Color functions
  - Browser type detection
  - URL functions
  - Document load/unload hooks
  - Effect functions for fading, sliding, and animating different style properties of an element

The standard Dojo profile (dojo.js) provided by the Ajax client runtime automatically loads the functionality above. See 12.2.4, “Creating a custom build” on page 566, for details of how to create customized versions of the dojo.js file.

- ▶ Core: Provides additional facilities on top of Dojo Base. These include:
  - Drag and drop support
  - Back-button support
  - String manipulation functions
  - RPC, IFrame communication
  - Internationalization (i18n)
  - Date
  - Number
  - Currency
  - Colors
  - Strings
  - Math
  - Data access
  - Regular expressions

- Debug facilities (via Firebug lite)
- Build system
- Markup parser
- OpenAjax hub 1.0

The Core library is described in 9.4, “The Dojo core library” on page 278.

- ▶ **Dijit:** A set of interaction rich widgets and themes for use when developing Ajax applications. Dojo widget features are:
  - Accessible (keyboard support, support for high-contrast mode, panel reader support)
  - High quality, neutral default theme (replaceable)
  - Extensive layout and form capabilities
  - Data-bound widgets
  - Grid and charts
  - Fully internationalized for many languages
  - Bidirectional (BiDi) support

The Dijit module is described in 9.7, “The Dojo widget library” on page 328. How to create custom widgets is discussed in 9.8, “Creating custom Dojo widgets” on page 350.

- ▶ **DojoX (Dojo eXtensions):** Contains features that may in the future migrate into Core, Dijit, or even a new module. Many of these extensions are community provided. See 9.10, “Dojo extensions in dojoX” on page 388, for an overview of current projects.
- ▶ **Util:** The Ajax client runtime provides the Dojo Objective Harness (DOH) in the Util module. DOH is a client testing framework. It is described in 9.11, “Unit testing with the Dojo Objective Harness” on page 390.

### 9.1.3 Dojo Foundation, licensing, and help

This section gives an overview of the Dojo Foundation, the licenses under which the toolkit can be used, and how you can find Dojo help.

#### The Dojo Foundation

The Dojo Toolkit is supported by the Dojo Foundation, which is a non-profit organization. For information regarding the foundation and other supported projects see:

<http://dojotoolkit.org/foundation>

## Licensing

Dojo is dual-licensed and can be used either under the terms of the Academic Free License Version 2.1 (AFL) or the Berkeley Software Distribution License. A detailed description of these licenses can be found at:

<http://trac.dojotoolkit.org/browser/dojo/trunk/LICENSE>

## Help

Dojo has a large active user and developer community. You can find help in forums, Inter Relay Chats (IRC), and mailing lists, as follows:

- Forums: Dojo's forum is available at:

<http://www.dojotoolkit.org/forum>

Before adding a new entry describing your problem, try first to find a solution on the Dojo search page at:

<http://www.dojotoolkit.org/search>

Additional help can be found on the FAQ page at:

<http://dojotoolkit.org/support/faq>

- IRC: As with any forums, it may take hours or days to receive an answer to your question. If you have an urgent question, try the #dojo chatroom on the irc.freenode.net server. Usually, some of the Dojo Toolkit developers are online and may give you a quick solution to your problem.
- Mailing list: If you prefer mailing lists, you can sign up for Dojo's main mailing list dojo-interest at:

<http://turtle.dojotoolkit.org/mailman/listinfo/dojo-interest>

- Other: Two other useful Web sites for Dojo news, code examples, and tutorials are:

<http://ajaxian.com/by/topic/dojo>

<http://dojocampus.org>

If you wish to participate in the Dojo Toolkit project as a contributor or committer, then read the contributor's license agreement at:

<http://www.dojotoolkit.org/community>

## 9.2 Ajax client runtime directory structure

This section describes the directory structure of the Ajax client runtime.

## 9.2.1 WebSphere 6.x servers

After extracting the WebSphere Feature Pack for Web 2.0 on WAS 6.x servers, the Ajax client runtime has the directory structure shown in Table 9-1.

Table 9-1 Ajax client runtime directory structure for WAS 6.x Multi-OS

Path to file	Content
<app_server_root>/Web2fep/Ajax-rt_1.X/	The non-optimized Ajax client runtime (the Dojo Toolkit Version 1.0 and extensions).
<app_server_root>/Web2fep/documentation/AjaxClientRuntime_1.X/dojo/index.html	The general user reference that links to the Dojo Toolkit Web site.
<app_server_root>/Web2fep/documentation/AjaxClientRuntime_1.X/ibm_atom/index.html	The general user reference for the Atom/APP extensions to the Dojo Toolkit.
<app_server_root>/Web2fep/documentation/AjaxClientRuntime_1.X/ibm_soap/index.html	The general user reference for the SOAP extensions to the Dojo Toolkit.
<app_server_root>/Web2fep/documentation/AjaxClientRuntime_1.X/ibm_opensearch/index.html	The general user reference for the OpenSearch extensions to the Dojo Toolkit.
<app_server_root>/Web2fep/documentation/AjaxClientRuntime_1.X/ibm_gauge/index.html	The general user reference for the Gauge extensions to the Dojo Toolkit.
<app_server_root>/Web2fep/site/eclipse-update-site.zip	An archived Eclipse site for use with Eclipse-based tools. The plug-ins contain development, as well as optimized profile builds of the Dojo Toolkit for use with application development.

## 9.2.2 WebSphere Server Community Edition and z/OS

For the WebSphere Server Community Edition and WebSphere 6.x servers on z/OS the directory structure of the Ajax client runtime after unzipping the WebSphere Application Server Feature Pack for Web 2.0 Zip-file is shown in Table 9-2.

Table 9-2 Ajax client runtime directory structure for WAS CE and z/OS

Path to file	Content
/documentation/dojo/index.html	General Dojo Toolkit documentation links
/documentation/ibm_atom/index.html	Atom extensions documentation
/documentation/ibm_soap/index.html	SOAP extensions documentation
/documentation/ibm_gauge/index.html	Gauges extensions documentation
/documentation/ibm_opensearch/index.html	OpenSearch extensions documentation
/dojo/	Dojo Core
/dijit/	Dijit Widgets
/dojox/	Dojo extensions
/ibm_soap/	SOAP extensions for the Dojo Toolkit
/ibm_atom/	Atom/APP extensions for the Dojo Toolkit
/ibm_opensearch/	OpenSearch extensions for the Dojo Toolkit
/ibm_gauge/	Gauge widget extensions for the Dojo Toolkit

## 9.3 Adding the Ajax client runtime to your project

This section describes how you can import the Ajax client runtime into a project and then include the Dojo Toolkit in your application.



### 9.3.1 Importing the libraries into the project folder

The Ajax client runtime directory can easily be imported into your Web application via the copy and paste function:

1. Open the directory of the Ajax client runtime.
  - For WAS 6.x MultiOS, this is `<app_server_root>/Web2fep/Ajax-rt_1.x`.
  - For the WebSphere Server Community Edition or on a z/OS machine, this is `AjaxClientRuntime_1.x` in the directory to which the Feature Pack for Web 2.0 Zip-file is extracted.
2. Copy the contents of the directory from step 1 and paste them into the WebContent folder in your project workspace in RAD or Eclipse. Figure 9-2 shows how the WebContent folder looks following the import.

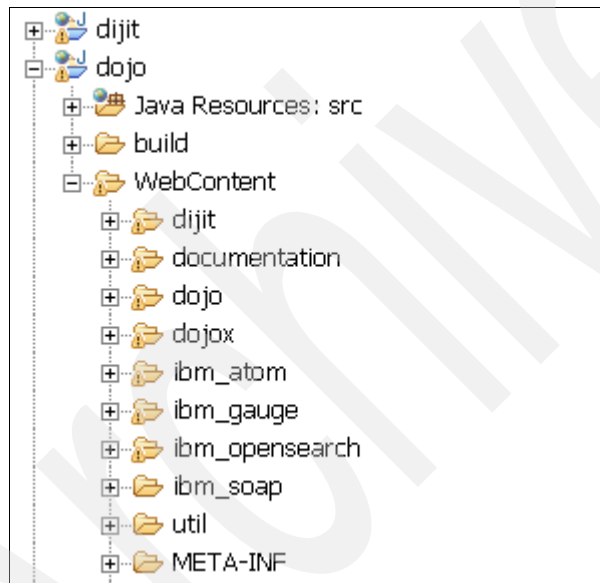


Figure 9-2 WebContent folder after Ajax client runtime import

If you prefer you can create a separate Dojo folder in the WebContent folder and copy the Ajax client runtime into that Dojo folder. The folder structure shown in Figure 9-2 is the one used for all of the examples contained in the following sections.

To verify that your dojo setup is correct, open `http://yourserver:port/dijit/themes/themeTester.html` and you should see the Dojo test page that gives an overview of the Dojo widgets.

### 9.3.2 Including the Dojo library

To extend your Web application with Dojo functionality, the first step is to include the main `dojo.js` file, which is under the `dojo` folder. For instance, this could be done in an `index.jsp` file that is used to access the application with the browser.

Presuming that your `index.jsp` is in the `WebContent` folder, Example 9-1 shows how to include the `dojo.js` file in your application.

*Example 9-1 Including the `dojo.js` file*

---

```
<html>
<head>
  <script type="text/javascript" src="dojo/dojo.js"></script>
</head>
<body>
</body>
</html>
```

---

## 9.4 The Dojo core library

This section describes the functionality provided by the Dojo core library and how it can be used to facilitate cross-browser JavaScript programming.

### 9.4.1 The `dojo` folder structure

The following gives a functionality overview of important subfolders and JavaScript files under the `dojo` folder.

- ▶ `_base`: This folder contains the Dojo base functionality for extend array processing, color code handling, event handling, simplified DOM tree programming, visual effects, JSON format handling, string handling, XHR communication, and a fast query engine for HTML elements. These `_base` features are discussed throughout the following sections.
- ▶ `data`: This folder contains Dojo's powerful Data API that enables you to populate elements with data in a simple way. The Data API is discussed in detail in 9.9, "Data API" on page 371.
- ▶ `dnd`: A framework to easily add drag and drop functionality to the application. Drag and drop is discussed in 9.4.9, "Drag and drop" on page 307.
- ▶ `rpc`: This folder provides functionality to call server-side functions directly with remote procedure calls and is described in 9.6.4, "Remote Procedure Calls" on page 327.

- ▶ `back.js`: An undesirable behavior introduced by the Ajax technology effects the use of the back button and the bookmarkability of Web pages. The `back.js` file helps minimize these issues in your Web application. See 9.4.10, “Handling the back-button problem” on page 308.
- ▶ `cookie.js`: Facilitates setting cookie values or getting cookie values. Its usage is described in 9.4.7, “Cookie handling” on page 301.
- ▶ `currency.js`, `date.js`, `i18n.js`: To deploy an application world wide, you may need to provide it in different languages. Dojo offers support to simplify internationalizing your Web application in an effective way. A full list of i18n-related steps that you must consider is provided in 12.5, “Internationalization (i18n)” on page 579.
- ▶ `dojo.js`: The main file that needs to be included in your Dojo application at least to use the base functionality. This file can be optimized for performance by creating your own custom build of Dojo.
- ▶ `OpenAjax.js`: The `OpenAjax.js` file contains a reference implementation of the OpenAjax Hub that can be used to integrate several Ajax-libraries in one JavaScript environment.
- ▶ `parser.js`: The Dojo parser is needed to parse Dojo proprietary Html tags and widgets. It is discussed in 9.8.5, “Using the Dojo parser” on page 361.

## 9.4.2 Dojo’s packaging system

JavaScript does not possess a packaging scheme. When the client-side code grows, it becomes difficult for the developer to maintain unique names for the functions and variables.

Dojo, however, provides a packaging scheme that allows for the creation of modules with specific functionality that resemble module organizations used in Java application packages. Unlike Java, a module can contain both function and class definitions.

The root Dojo namespace after including the `dojo.js` file in your application is *dojo* itself. Any function of the Dojo core libraries is preceded by `dojo`, for example, `dojo.byId()`.

### **dojo.require and dojo.provide**

The two important functions of the Dojo packaging system are `dojo.provide(String moduleName)` and `dojo.require(String moduleName)`.

The `dojo.provide` statement should be the first line of each JavaScript file that you include in the application. It defines the module name of your JavaScript file and can be compared with the package command in Java. `dojo.provide` takes one

parameter that is a unique name across the entire application. The name is the relative path from the directory to the file including the file name. For instance, if you have a directory myPackage and a myModule.js inside and myPackage is located under the Dojo root directory (in this case, it is WebContent) at the same level as the dojo folder, then you would add `dojo.provide("myPackage.myModule")` as the first line in myModule.js. A dot is used as the separator and no .js extension should be added.

The `dojo.require` function is used to include the functionality of other JavaScript files and can be compared with Java's import command. This means that you do not need to use script tags to include those files as long as other JavaScript files contain a `dojo.provide` statement. The benefit of using the `dojo.require` function is that before loading the module the function first checks whether the module is already loaded. If it is, it does not attempt to load it again. If there are other `dojo.require` statements contained in a JavaScript file that was previously loaded via a `dojo.require` statement, then all dependent JavaScript files are also loaded. This keeps the developer from having to provide a similar mechanism.

A typical JavaScript file using `dojo.require` is shown in Example 9-2.

*Example 9-2 Usage of dojo.require*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"></script>
<script type="text/javascript">
    dojo.require("dojo.back"); //path is dojo/back
    dojo.require("dojo.date.locale"); //path is dojo/date/locale
</script>
<body>
</body>
</html>
```

---

## Defining a module path

Dojo presumes that all file paths are relative to the dojoRoot/dojo directory. You should create your directories at the same level as the dojo directory. For instance, examine the directory structure shown in Figure 9-3.

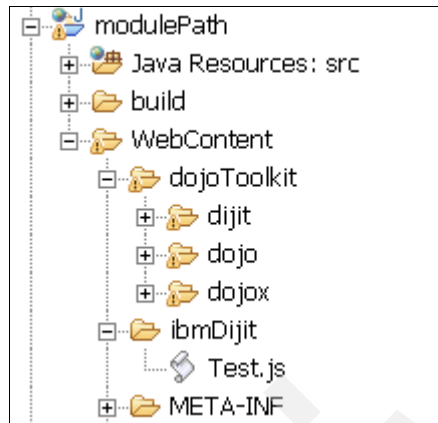


Figure 9-3 Different Dojo folder structure

If you were to perform a `dojo.require("ibmDijit.Test")` then Dojo will throw an error since it will fail to locate the `Test.js` file in the relative path `dojoToolkit/ibmDijit/`. To inform Dojo that a file is placed outside of the Dojo root folder, you can include the `dojo.registerModulePath` statement as shown in Example 9-3.

Example 9-3 Registering of a module path

```
<html>
<script type="text/javascript" src="dojoToolkit/dojo/dojo.js"></script>
<script type="text/javascript">
    // file is under ../../ibmDijit relative to dojoToolkit/dojo
    dojo.registerModulePath("ibmDijit", "../../ibmDijit")
    dojo.require("ibmDijit.Test"); //path is dojo/back

    dojo.addOnLoad(function(){
        var ref = dojo.moduleUrl("ibmDijit","templates/images/Test.gif");
        dojo.byId("image").src = ref;
    });
</script>
<body>
<img id="image">
</body>
</html>
```

The previous example shows how you can access resources relative to a specific folder if a module path is registered. However, if you are using the `dojo.moduleUrl` statement and you might move folders within the project, then you only need to specify the new path location using the `dojo.registerModulePath` statement and no change is needed for the `dojo.moduleUrl` calls. Otherwise, you can just use the relative path to locate the file. For instance, if you want to use `Test.gif` in `Test.js`, you can easily access it with `image.src="templates/images/Test.gif"` instead of using the `dojo.moduleUrl` statement.

Another method when using the `registerModulePath` statement is to set the `djConfig` parameter `modulePaths`, which is described in the following section.

### 9.4.3 Dojo configuration with `djConfig`

You can set the global object `djConfig` to enable or disable specific Dojo features. The `djConfig` statement must be defined before the `dojo.js` file is included. You can do that in a separate script tag, as shown in Example 9-4, or set the attribute `djConfig` in the script tag that includes the `dojo.js` file, as shown in Example 9-5.

*Example 9-4 Setting `djConfig` in a script tag*

---

```
<html>
<script type=text/javascript>
  var djConfig = { parseOnLoad:true, isDebug: true };
</script>
<script type="text/javascript" src="dojo/dojo.js"></script>
<body>
</body>
</html>
```

---

*Example 9-5 Setting `djConfig` as an HTML attribute*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="parseOnLoad:true, isDebug: true"></script>
<body>
</body>
</html>
```

---

A list of important djConfig parameters you can set are listed below.

- ▶ **baseUrl**: If you rename `dojo.js`, set `baseUrl` to the folder that contains the renamed file, for example, `dojo/`. Usually, you do not need to set it. Dojo auto-populates `baseUrl`.
- ▶ **cacheBust**: If set, Dojo adds the value of `cacheBust` to the URL that loads a resource such as a JavaScript module. For instance, if you set `cacheBust` to `(new Date()).getTime()`, the first time that you start the application a module could be loaded with `http://server:port/path/module.js?1210337274094` and the second time with `http://server:port/path/module.js?2310837536723`. Both load the same resource, but because the parameter list is different the browser is forced to load it from the server and does not take it from the cache. This can be useful during module development.
- ▶ **debugAtAllCosts**: The default is `false`. If `true`, it can be used to debug JavaScript modules at the cost of performance. Since it slows down the client and has some unpredictable side effects you should only use it during the development stage and not in the production code.
- ▶ **isDebug**, **debugContainerId**, **debugHeight**, **noFirebugLite**, **popup**: Firebug is the tool for debugging in the Firefox browser. It provides `console.*` functions such as `console.debug` to log messages in the Firebug window that is installed as a plug-in for the Firefox browser.

**FirebugLite** (<http://www.getfirebug.com/lite.html>) provides debugging facilities similar to Firebug for other browsers like Internet Explorer, Opera, and so on.

The Dojo Toolkit includes the **FirebugLite** library, which can be used with the `isDebug`, `debugContainerId`, `debugHeight`, `noFirebugLite`, and `popup` parameters.

- **Firefox**: If the Firebug plug-in is enabled, then the `console.*` function calls are always displayed in the Firebug console window. It does not matter if any of the `djConfig` debugging parameters are set. If the Firebug plug-in is disabled or not installed then the following statements for other browsers apply.
- **Other browsers**: When using Dojo, `console.*` functions can be called from all browsers without causing an error. Dependent on the following parameters, log messages are displayed or not.
  - **isDebug**: The default `false`. If `true`, then Dojo adds a console window after the last HTML element on the page. Calls to `console.*` will output to that window.
  - **debugContainerId**: If set to an ID of an HTML element on the page (for example, a `div` with a specific size and position on the page), then the

console window is inserted into that element instead of appending it after the last element on the page.

- `debugHeight`: The height of the console window.
- `popup`: The default is false. If true, the console window opens in a new pop-up, ignoring `debugContainerId` and `debugHeight`.
- ▶ `dojoBlankHtmlUrl`, `dojoIframeHistoryUrl`, `useXDomain`: When using a cross-domain Dojo build, set the `useXDomain` value to true. Save the `dojo/resources/blank.html` and `dojo/resources/iframe_history.html` to your domain and set the `dojoBlankHtmlUrl` and the `dojoIframeHistoryUrl` to the path where you saved the files in your domain.
- ▶ `locale`, `extraLocale`: Needed for internationalization (i18n) of displayed strings numbers, date, currencies, and so on. See 12.5.1, “Dojo i18n” on page 580, for more details. If its not set, the default value is `navigator.userLanguage` or `navigator.language`.
- ▶ `modulePaths`: A map for setting module paths relative to `djConfig.baseUrl`. It can be used as an alternative to `dojo.registerModulePath`. Simply pass a map with the module path like `modulePaths`:  

```
{ "ibmDijit": "../..//ibmDijit", "ibmDijit2": "../..//ibmDijit2" }
```
- ▶ `parseOnLoad`: If set to true, Dojo automatically parses the `HTMLDocument` when the application is started and replaces Dojo-specific code in the markup with the normal HTML markup. The default is false, which means that if you use widgets in your markup you have to parse the document manually. Parsing is discussed in 9.8.5, “Using the Dojo parser” on page 361.
- ▶ `preventBackButtonFix`: Set this to false when using Dojo’s back-button module. Back-button functionality is discussed in 9.4.10, “Handling the back-button problem” on page 308.
- ▶ `require`: Instead of using `dojo.require` in a script tag and load modules separately, you can pass all module names in an array to `djConfig.require` and Dojo loads them automatically. For instance, `require`:  

```
[ "dojo.back", "dojo.date", "dijit.form.Button" ]
```
- ▶ `usePlainJson`: If you use the value `json` or `json-comment-optional` as the `handleAs` property in a `dojo.xhr` call then Dojo outputs ‘Consider using `mimetype:text/json-comment-filtered` to avoid potential security issues ...’ in the Firebug/Lite window. If you wish to avoid that message, set `usePlainJson` to true. The default value is false. The `dojo.xhr` functions are discussed in 9.6.1, “XMLHttpRequest” on page 316.

In most situations you only need to use `isDebug`, `parseOnLoad`, and `locale`, and allow Dojo to assign default values for the others.



## 9.4.4 Useful Dojo core functions

This section gives an overview of handy functions that you do not want to miss when programming with Dojo. The functions are:

- ▶ `dojo.require`: This is used to load a module and can be compared with Java's `import` command. A module is always only loaded once, even if there are several `dojo.require` calls for the same module. If the module does not exist an error like `Could not load module name` is thrown. Example 9-6 shows the usage.

*Example 9-6 Usage of `dojo.require`*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"></script>
<script type="text/javascript">
    dojo.require("dijit.form.Button");
    dojo.require("dojo.parser");
    dojo.require("yourmodule.modulename");
</script>
<body>
</body>
</html>
```

---

A good position in which to place the `dojo.require` calls is directly following the `dojo.js` statement.

- ▶ `dojo.byId`, `dijit.byId`: The function `dojo.byId(String id)` returns a reference to the HTML node of an element `dijit.byId(String id)`, a reference to a widget. `dojo.byId` is a shorter version of `document.getElementById`. Example 9-7 shows the usage.

*Example 9-7 Usage of `dojo.byId` and `dijit.byId`*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
    djConfig="parseOnLoad: true"></script>
<script type="text/javascript">
    dojo.require("dijit.form.Button");
    dojo.require("dojo.parser");
</script>
<script type="text/javascript">
    dojo.addOnLoad(function() {
        var ref = null;

        // ref = reference to the div with id 'htmlNode'
        ref = dojo.byId("htmlNode");
```

```

// call any function you can call on an Html node
ref.style.background = "yellow";

// ref = reference to the domNode of the widget with id 'btn'
ref = dojo.byId("btn");
// call any function you can call on an Html node
ref.style.background = "red";

// ref = reference to the dijit with id 'btn'
ref = dijit.byId("btn");
// call any function that is defined in the widget class
ref.setLabel("dijitById");
});
</script>
<body>
  <div dojoType="dijit.form.Button" id="btn">Button</div>
  <div id="htmlNode">This is an Html node</div>
</body>
</html>

```

- 
- ▶ `dojo.forEach`, `dojo.indexOf`, `dojo.lastIndexOf`, `dojo.every`, `dojo.some`, `dojo.filter`, `dojo.map`: Dojo provides cross-browser versions of these array functions, as some browsers like Internet Explorer do not support them. Example 9-8 shows how to use each of these functions.

---

*Example 9-8 Usage of Dojo's array functions*

---

```

<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="isDebug: true"></script>
<script type="text/javascript">
  var arr = [1, 2, 3, 4];

  // runs the function for each element in the array
  // if the index is not needed, use only element as parameter
  dojo.forEach(arr, function(element, index) {
    console.debug("index: " + index + ", val: " + element);
    // index: 0, val: 1
    // index: 1, val: 2
    // index: 2, val: 3
    // index: 3, val: 4
  });

  // returns true if the passed function returns true for each of the
  // elements

```

```

bool = dojo.every(arr, function(element) {
    // check if element is not a number
    return element > 2;
});
// bool = false

// returns true if the passed function returns true for at least one
// element
bool = dojo.some(arr, function(element) {
    return element > 2;
});
// bool = true

// return index of element, start searching at index 0, returns -1
// when not found
index = dojo.indexOf([1, 2, 3, 2], 2);
// index = 1

// return index of element, start searching at end of array
index = dojo.lastIndexOf([1, 2, 3, 2], 2);
// index = 3

// apply passed function to each element and return an array with
// the returned values of that function
res = dojo.map(arr, function(element) {
    return element * element;
});
// res = [1, 4, 9, 16]

// filters array elements depending on the return value of the
// passed function
res = dojo.filter(arr, function(element) {
    return element < 3;
});
// res = [1, 2]
</script>
<body>
</body>
</html>

```

---

- ▶ `dojo.addOnLoad`: This runs commands after the HTML body is loaded and Dojo initialization has completed. If you use the `onload` attribute in the body tag like `<body onload="someFunc">` it is not guaranteed that the Dojo initialization has completed. Use `dojo.addOnLoad` instead. Example 9-9 shows the usage.

*Example 9-9 Usage of `dojo.addOnLoad`*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"></script>
<script type="text/javascript">
    dojo.addOnLoad(function() {
        // body loaded and Dojo initialized
    });
</script>
<body>
</body>
</html>
```

---

- ▶ `dojo.hitch`: A function that executes within a given scope in the format `dojo.hitch(scope, function)`. This function is most useful when you need to access the reference inside a widget class, as shown in Example 9-10.

*Example 9-10 Usage of the `dojo.hitch` function*

---

```
dojo.declare(
    "ibmDijit.Test", [dijit._Widget, dijit._Templated],
    {
        templateString: "<div><button dojoAttachPoint='btn'>Test
dojo.hitch</button></div>",

        testHitch: function() {
            console.debug("dojo.hitch rocks!")
        },

        postCreate: function(){
            dojo.connect(this.btn, "onclick", function(){
                // this reference cannot be accessed in this scope
                // -> 'this.testHitch is not a function' thrown
                this.testHitch();
            });

            dojo.connect(this.btn, "onclick", dojo.hitch(this, function(){
                // works fine
                this.testHitch();
            }));
        }
    }
);
```

```

dojo.subscribe("/testHitch", dojo.hitch(this, function() {
    // works fine
    this.testHitch();
}));

dojo.publish("/testHitch");

dojo.xhrGet ({
    url: '/testHitch.html',
    load: dojo.hitch(this, function (data) {
        // this reference can be accessed inside;
        this.testHitch();
    }),
    // Called if there was an error (such as a 404 response)
    error: dojo.hitch(this, function (data) {
        // this reference can be accessed inside;
        this.testHitch();
    })
});
}
})

```

- ▶ **dojo.destroyRecursively:** To get rid of a widget and all of its children on your page, use `dojo.destroyRecursively`.
- ▶ **dojo.style:** This is used to set or get style attributes of an HTML node. Example 9-11 shows the usage.

---

*Example 9-11 Usage of `dojo.style`*

---

```

<html>
<script type="text/javascript" src="dojo/dojo.js"></script>
<script type="text/javascript">
    dojo.addOnLoad(function() {
        var testDiv = dojo.byId("testDiv");

        // if third argument passed, it acts as a setter
        dojo.style(testDiv, "width", "500px");

        // if a string is passed as first argument, Dojo automatically
        // searches for the node with that id
        dojo.style("testDiv", "background", "yellow");

        //if no third argument passed, it acts as a getter
        res = dojo.style("testDiv", "width");
    });

```

```

        // res = 500px

        res = dojo.style("testDiv");
        // res = object with ALL styles that are set manually or computed
        // by the browser
        // {}
    });
</script>

<body>
    <div id="testDiv" >Test dojo.style</div>
</body>
</html>

```

---

In Dojo 1.0 it is only possible to set one style attribute in a single call. In Dojo 1.1 you can pass an object with several key-value pairs to set more than one style, for example, `dojo.style(yourNode, {height:"100px", width: "100px"})`.

In 9.4.11, “Selecting DOM nodes with Dojo’s query engine” on page 312, we describe how the style function is used together with the `dojo.query` function.

- ▶ `dojo.hasClass`, `dojo.addClass`, `dojo.removeClass`, `dojo.toggleClass`: Functions to check whether a node has a specific styling class, to add or remove a class, and to toggle between classes of a node. Example 9-12 shows their usage.

---

*Example 9-12 Adding class attributes with Dojo*

---

```

<html>
<script type="text/javascript" src="dojo/dojo.js"></script>
<script type="text/javascript">
    dojo.addOnLoad(function() {

        // pass the actual node reference or the id of the node
        res = dojo.hasClass("testDiv", "class1");
        // res = true

        dojo.addClass("testDiv", "class2");
        // class = "class1 class2"

        dojo.removeClass("testDiv", "class1");
        // class = "class2"

        dojo.toggleClass("testDiv", "class1");
        // class = "class2 class1"

        dojo.toggleClass("testDiv", "class1");
    });

```

```

        // class = "class2"
    });
</script>

<body>
    <div id="testDiv" class="class1">Test dojo.style</div>
</body>
</html>

```

---

- ▶ **dojo.toJson, dojo.fromJson:** The function `dojo.toJson` is used to serialize a JSON object into a string. The `dojo.fromJson` is used to deserialize a string into a JSON object, as shown in Example 9-13.

---

*Example 9-13 Serializing from and deserializing to a JSON object*

---

```

<html>
<script type="text/javascript" src="dojo/dojo.js"></script>
<script type="text/javascript">
    var jsonString = "{key1: 'val1', key2: [1,'two']}";

    res = dojo.fromJson(jsonString).key1;
    // res = val1

    res = dojo.toJson({key1: "val1", key2: [1,"two"]});
    // res = {"key1": "val1", "key2": [1, "two"]}
</script>
<body>
</body>
</html>

```

---

- ▶ **dojo.connect:** This is used to connect DOM-events to function calls or function calls to other function calls. See 9.4.8, “Dojo’s event system” on page 302, for more details.
- ▶ **dojo.subscribe, dojo.publish:** Used to publish events and subscribe to events. See 9.5, “Dojo’s pub/sub event-system” on page 313, for more details.
- ▶ **dojo.trim, dojo.string.substitute:** `dojo.trim(String string)` removes left and right paddings in a string. The function `dojo.string.substitute` is used to substitute placeholders in a string and is described in 12.5, “Internationalization (i18n)” on page 579.
- ▶ **dojo.isFunction, dojo.isArray, dojo.isObject:** These functions return true if the passed argument is of the type the function is checking for, for example, `dojo.isObject({})` returns true.
- ▶ **dojo.xhrGet, dojo.xhrPost:** These functions are a cross-browser abstraction of the XMLHttpRequest object of the browser and are used for asynchronous

communication with the server. See 9.6.1, “XMLHttpRequest” on page 316, for more details.

- ▶ `dojo.query`: Invokes the powerful Dojo query engine. This is described in 9.4.11, “Selecting DOM nodes with Dojo’s query engine” on page 312.

## 9.4.5 Object-oriented programming with Dojo

JavaScript allows object-oriented programming with the `prototype` attribute. However, the use of the `prototype` attribute is not easy and makes it difficult for inexperienced JavaScript programmers to efficiently develop classes. Dojo provides its own class system to create classes and implement inheritance in a more abstract way.

**Note:** JavaScript does not have access-level modifiers. It is the developer’s responsibility to visually define which classes, functions, or variables are to be public or private. For instance, all private functions and variables can be preceded by an underscore (`_`). This is also the general Dojo convention that is used throughout the toolkit.

### Defining a class and creating an instance

Example 9-14 shows the basic structure of a Dojo class defined with the `dojo.declare` function.

*Example 9-14 Basic structure of a Dojo class*

```
dojo.provide("ibm.Class");
dojo.declare(
    "ibm.Class", null,
    {
        var1: 0,
        var2: "",
        _privateVar: "",

        constructor: function(param1, param2) {
            this.param1 = param1;
            this.param2 = param2;
        }
    }
);
```



```
        this._privateVar = "privateVar";  
    },
```

```
increaseVar
```

---

The function `dojo.declare` has the following parameters:

- ▶ first param: This is the name of the class. It has the same structure as a module name as described in 9.4.2, “Dojo’s packaging system” on page 279.
- ▶ second param: One class or an array that contains the classes to inherit from. Pass null if you do not want to extend another class.
- ▶ third param: This is the real body of the class. It is an object with key-value-mappings. Values can be anything, for example, strings, numbers, object references, anonymous functions, and so on.

The *this* reference points to the object itself as it does in Java. Note that JavaScript allows dynamic creation of variables during run time. The variable `param1` in the example above is accessible in every other function in the class after it is assigned the value in the constructor. However, adding all needed variables as a key in the third parameter for `dojo.declare` (like `var1`) gives a better overview of the class and may make it easier to track problems for large classes.

The *constructor* function has a special role. It is called by Dojo automatically each time a new instance of the object is created, as shown in Example 9-15. Since JavaScript does not allow method overloading, you can only have one class constructor.

*Example 9-15 Instantiating a Dojo class*

---

```
dojo.require("ibm.Class");  
var test = new ibm.Class("param1", "param2");  
test.increaseVar1(); // var1 = 1
```

---

## Class and object variables

Variables can be passed as key-value pairs or defined in the functions like the constructor. If you pass them as key-value pairs, then variables with simple types like strings or numbers are not static. However, non-simple variables pointing to arrays or referencing other objects are static and are shared among all created instances of the class. If you define non-simple variables or populate a non-simple variable inside the constructor function, then each object of the appropriate class will have its own copy of those variables.

### *Example 9-16 Static variables*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="isDebug:true"></script>
<script type="text/javascript">
    // no need for dojo.provide or dojo.require for the class if you
    // define the class on your starting page
    dojo.declare("ibm.Class", null, {
        var1: 0,
        var2: "val",
        var3: {key: "val"},
        var4: {key: "val"}, // static
        var5: [1, 2, 3], // static

        constructor: function() {
            // when populated in the constructor each instance has a copy
            this.var3 = { key: "val1" };
        }
    });

    var class1 = new ibm.Class();
    var class2 = new ibm.Class();

    class1.var1 = 10;
    console.debug(class2.var1); // 0

    class1.var2 = "class1";
    console.debug(class2.var2); // val

    class1.var3.key = "class1";
    console.debug(class2.var3.key); // val1

    class1.var4.key = "class1";
    console.debug(class2.var4.key); // class1
```

```
class1.var5[0] = 10;
console.debug(class2.var5[0]); // 10
</script>
<body>
</body>
</html>
```

---

## Inheritance

Inheritance is specified by the second parameter of the `dojo.declare` function. By passing only the name of one class you can simulate inheritance as it is implemented in Java.

### *Example 9-17 Inheritance in Dojo classes*

```
dojo.provide("ibm.SubClass");
dojo.require("ibm.Class");
dojo.declare("ibm.SubClass", ibm.Class, {
    // variables, constructor, functions
});
```

---

## Multiple inheritance with mixins

Dojo allows multiple inheritance by passing an array of class names as the second parameter to the `dojo.declare` function. In terms of Dojo, only the first element in the array is the real superclass of the inheriting subclass. The other elements are called mixins. When using multiple inheritance the constructor of the superclass is called first, then the constructors of the mixins in order of definition in the array, and then the constructor of the child. Example 9-18 shows multiple inheritance with mixins.

### *Example 9-18 Multiple inheritance with mixins*

```
dojo.provide("ibm.SubClass");
dojo.require("ibm.Class");
dojo.provide("ibm.MixinClass");
dojo.declare("ibm.SubClass", [ibm.Class, ibm.MixinClass], {
    // variables, constructor, functions
});
```

---

## Overriding functions

The constructors of the superclass, mixins, and of the child that is instantiated are called automatically by Dojo. If you override a function of the parent class in the child and wish to call the parent function use the function `this.inherited` as it is shown in Example 9-19.

*Example 9-19 Overriding functions*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="isDebug:true"></script>
<script type="text/javascript">
    dojo.require("ibm.SubClass");
</script>
<script type="text/javascript">
    dojo.declare("ibm.Class", null, {
        test: function(param1, param2) {
            console.debug(param1);
            console.debug(param2);
        }
    });

    dojo.declare("ibm.SubClass", ibm.Class, {
        test: function(param) {
            this.inherited(arguments);
            // do something
        }
    });

    var subclass = new ibm.SubClass();
    subclass.test("param1", "param2");
</script>
<body>
</body>
</html>
```

---

The arguments parameter is a special JavaScript array and holds all arguments passed to the overriding function itself.

### 9.4.6 Visual effects

A major feature of Ajax programming is that following a user interaction only the needed parts of the Web page are refreshed asynchronously and not the whole page. However, it might be difficult for the user to notice small changes occurring

on the panel. Visual effects can be used both to notify the user about changes and also to make the application more visually appealing.

## Base effects

The effect functions that come with Dojo's base functionality are the `dojo.fadeIn`, the `dojo.fadeOut`, and the `dojo.animateProperty`. Example 9-20 shows how to use `dojo.fadeIn` and `dojo.fadeOut`.

*Example 9-20 Dojo base effects*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="isDebug:true"></script>
<script type="text/javascript">
    function effectFadeOut() {
        var effect = dojo.fadeOut({
            node: "fadeOut",
            // fading out shall take 2 seconds
            duration: 2000,
            // 10 frames per second
            rate: 10,
            // start animation after 1 second
            delay: 1000,
            // fired by Dojo
            beforeBegin: function(){ console.debug("beforeBegin"); },
            // fired by Dojo
            onBegin: function(){ console.debug("onBegin"); },
            // fired by Dojo
            onAnimate: function() { console.debug("onAnimate"); },
            // fired manually by calling effect.play()
            onPlay: function() { console.debug("onPlay"); },
            // fired by Dojo
            onEnd: function() { console.debug("onEnd"); },
            // fired manually by calling effect.pause()
            onPause: function() { console.debug("onPause"); },
            // fired manually by calling effect.stop()
            onStop: function() { console.debug("onStop"); }
        });
        // start animation
        effect.play();
        // effect status either paused, playing, or stopped
        console.debug("status: " + effect.status());
        // stop the animation manually
        effect.stop();
        console.debug("status: " + effect.status());
    }
</script>
```

```

        effect.play();
        effect.pause();
        console.debug("status: " + effect.status());
        // set the progress of the animation, if second param true, then
        // the animation is played automatically, otherwise effect.play()
        // needed
        effect.gotoPercent(0.5, true);
        console.debug("status: " + effect.status());
    };

    function fadeOutIn() {
        dojo.fadeOut({
            node: "fadeOutIn",
            duration: 1000,
            onEnd: function() {
                dojo.fadeIn({
                    node: "fadeOutIn",
                    duration: 1000
                }).play();
            }
        }).play();
    };

    dojo.addOnLoad(function() {
        effectFadeOut();
        fadeOutIn();
    });
</script>
<body>
    <div id="fadeOut">This text is faded out</div>
    <div id="fadeOutIn">This text is faded out and in</div>
</body>
</html>

```

---

In most situations it is sufficient to pass only the node and duration properties (as they are shown in the fadeOutIn function) and allow Dojo to assign the default values for the other properties.

The function `dojo.animateProperty` is used to animate CSS-property values of an HTML-node. Example 9-21 shows the usage.

*Example 9-21 Usage of `dojo.animateProperty`*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="isDebug:true"></script>
<script type="text/javascript">
    dojo.addOnLoad(function() {
        dojo.animateProperty({
            node: "animProp",
            duration: 2000,
            onEnd: function() { /*...*/ },
            properties: {
                width: {start: "0", end: "300", unit: "px"},
                height: {start: "0", end: "200"},
                backgroundColor: {start: "#111111", end: "#aaaaaa"}
            }
        }).play();
    });
</script>
<body>
    <div id="animProp">This div is animated</div>
</body>
</html>
```

---

Simply pass the CSS-properties that you wish to animate to the properties object and Dojo will animate the property from the start value to the end value. If you do not pass the unit property the default unit size for position or size animation is pixel. As shown previously for the `dojo.fadeOut` function, you can also hook into all the calls like `onEnd`, `onPlay`, and so on.

**Note:** All effects are played asynchronously and do not block.

### More effect features in `dojo.fx`

If you included the `dojo.fx` module by specifying `dojo.require("dojo.fx")` you can use the following additional effect features:

- ▶ `dojo.fx.wipeIn`, `dojo.fx.wipeOut`: Usage is similar to `dojo.fadeIn` and `dojo.fadeOut`.
- ▶ `dojo.fx.slideTo`: Enables you to move a node from its current position to another position on the page. Usage: `dojo.fx.slideTo({ node: node, left:"40",`

top:"50", unit:"px" })).play(). Additionally, you can pass the properties onEnd, onPlay, and so on, as you do for dojo.fadeIn.

- ▶ `dojo.fx.combine`, `dojo.fx.chain`: Starts several effects at the same time or runs the effects in a specific sequence. Example 9-22 shows the usage. Note that you have to use `dojo.connect` to hook into the anim cycle, as the functions only take an effect array as the argument. The function `dojo.connect` is described in 9.4.8, “Dojo’s event system” on page 302.

*Example 9-22 Combining and chaining effects*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="isDebug:true"></script>
<script type="text/javascript">
    dojo.require("dojo.fx");
</script>
<script type="text/javascript">
    dojo.addOnLoad(function() {
        var effects = [dojo.fadeOut({ node: "div1" }),
                        dojo.fadeOut({ node: "div2" })];
        var anim = dojo.fx.chain(effects);
        // var anim = dojo.fx.combine(effects);

        // use dojo.connect to hook into the anim cycle
        dojo.connect(anim, "onPlay", function() {
            console.debug("anim started");
        });
        dojo.connect(anim, "onEnd", function() {
            console.debug("anim ended");
        });

        anim.play();
    });
</script>
<body>
    <div id="div1">This div is animated</div>
    <div id="div2">This div is animated</div>
</body>
</html>
```

---



- `dojo.fx.Toggler`: Used to toggle an element between states. The usage is shown in Example 9-23.

*Example 9-23 Toggling effects with `dojo.fx.Toggler`*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="isDebug:true"></script>
<script type="text/javascript">
    dojo.require("dojo.fx");
</script>
<script type="text/javascript">
    dojo.addOnLoad(function() {
        var toggler = new dojo.fx.Toggler({
            node: "toggle",
            showFunc: dojo.fx.wipeIn, //default dojo.fadeIn
            showDuration: 1000, //default 200
            hideFunc: dojo.fx.wipeOut, //default dojo.fadeOut
            hideDuration: 1000 //default 200
        });

        toggler.hide(); //calls showFunc
        // ...
        // toggler.show(); //calls hideFunc
    });
</script>
<body>
    <div id="toggle">This div is animated</div>
</body>
</html>
```

---

## 9.4.7 Cookie handling

Dojo provides a function `dojo.cookie` for cookie handling in the browser. Example 9-24 shows how to set a cookie value, how to read a value of a cookie, and how to delete it.

*Example 9-24 Cookie handling with Dojo*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="isDebug:true"></script>
<script type="text/javascript">
    dojo.require("dojo.cookie");
</script>
```

```
<script type=text/javascript>
  // sets myCookie with the value 'someValue'
  dojo.cookie("myCookie1", "someValue");
  // use JSON to save complex data
  dojo.cookie("myCookie2", dojo.toJson({key:"value"}));
  dojo.cookie("myCookie3", "someValue", {expires: 10});

  // getting cookie values, only pass cookie name
  var myCookie1 = dojo.cookie("myCookie1");
  var myCookie2 = dojo.fromJson(dojo.cookie("myCookie2"));

  // delete cookie with expires -1 or setting to null
  dojo.cookie("myCookie1", "", {expires: -1});
  dojo.cookie("myCookie2", null);
</script>
<body>
</body>
</html>
```

---

The third parameter can have the following properties:

- ▶ **expires:** If a number is passed, then the cookie expires after that number of days from the time the cookie is created. You can also pass a JavaScript Date object to specify the date that the cookie will become invalid. If set to a -1 or a date in the past, the cookie is deleted. If it is set to 0 or not passed at all, the cookie is deleted when the browser closes.
- ▶ **path:** Specifies the path where the cookie is active. For example, use a slash (/) if the cookie is valid throughout the entire domain.
- ▶ **domain:** The domain to which the cookie is sent. If not specified, the cookie is sent to the domain of the page that the cookie is set in.
- ▶ **secure:** Specifies whether the cookie is sent on a secure connection.

### 9.4.8 Dojo's event system

Dojo comes with its own cross-browser event system that not only allows for event handlers to be attached to DOM-elements, such as button-clicks, but also allows a function call to be connected to other function calls. It is also possible to subscribe to a topic and publish topics, as discussed in 9.5, “Dojo's pub/sub event-system” on page 313.

## Attaching event-handlers with dojo.connect

The `dojo.connect` function is provided by the Dojo base functionality. Once you have included the `dojo.js` file you can start using the event system to capture events that you do not want your Web application to miss.

Example 9-25 shows the signature of the `dojo.connect` function.

*Example 9-25 Signature of the `dojo.connect` function*

---

```
dojo.connect(/*Object*/ obj, /*String*/ event, /*Object|null*/ context,  
            /*String|Function*/ method, /*Boolean*/ dontFix)
```

---

The parameters are:

- ▶ **obj**: This is the source object that fires the event. If it is null then Dojo takes `dojo.global` as the source. `dojo.global` is a reference to the global scope in the browser. `obj` can be a DOM-node or another normal object reference.
- ▶ **event**: The event that is fired on the source `obj`. This can be a normal DOM-event or a function that is called on `obj`.
- ▶ **context**: The context in which the method is invoked. This can be either a reference to an object or `dojo.global` if set to null.
- ▶ **method**: The method that is called when the event is fired. It receives the arguments that are passed to the event.
- ▶ **dontFix**: If `dontFix` is set to true and `obj` is a DOM-node, the delegation of this connection to the DOM event manager is prevented.

## Connecting events

Example 9-26 shows how to use the `dojo.connect` function to connect DOM-events to function calls or connect one function call to another.

*Example 9-26 Usage of the `dojo.connect` function*

---

```
<html>  
<script type="text/javascript" src="dojo/dojo.js"  
djConfig="isDebug:true"></script>  
<script type="text/javascript">  
    function fireEvent() {  
        // do something  
    };  
    function testEvent() {  
        console.debug("dojo.connect global");  
    };  
  
    // instead of a normal object, you can use a reference to a
```

```

// class instance or widget instance in the same way as for object
var object = {
    fireEvent: function() { /*do something*/ },
    testEvent: function() { console.debug("dojo.connect object"); }
};

// dojo.connect calls in the same block have the same result

// connect DOM-node to object function
dojo.connect(dojo.byId("btn"), "onclick", object, "testEvent");
dojo.connect(dojo.byId("btn"), "onclick", object, object.testEvent);
// when button clicked, prints 2 times 'dojo.connect object'

// connect DOM-node to global function
dojo.connect(dojo.byId("btn"), "onclick", null, testEvent);
dojo.connect(dojo.byId("btn"), "onclick", null, "testEvent");
dojo.connect(dojo.byId("btn"), "onclick", "testEvent");
dojo.connect(dojo.byId("btn"), "onclick", testEvent);
// when button clicked, prints 4 times 'dojo.connect global'

// connect global function to global function
dojo.connect(null, "fireEvent", null, "testEvent");
dojo.connect(null, "fireEvent", null, testEvent);
dojo.connect(null, "fireEvent", "testEvent");
dojo.connect("fireEvent", testEvent);
dojo.connect("fireEvent", "testEvent");
fireEvent();//prints 5 times 'dojo.connect global'

// connect object function to object function
dojo.connect(object, "fireEvent", object, "testEvent");
dojo.connect(object, "fireEvent", object, object.testEvent);
object.fireEvent();//prints 2 times 'dojo.connect object'

// connect global function to object function
dojo.connect(null, "fireEvent", object, "testEvent");
dojo.connect("fireEvent", object, "testEvent");
dojo.connect("fireEvent", object, object.testEvent);
fireEvent();//prints 3 times 'dojo.connect object'

// connect object function to global function
dojo.connect(object, "fireEvent", null, "testEvent");
dojo.connect(object, "fireEvent", null, testEvent);
dojo.connect(object, "fireEvent", testEvent);
dojo.connect(object, "fireEvent", "testEvent");
object.fireEvent();//prints 4 times 'dojo.connect global'

```

```
// for the following dojo.connect call you can replace the obj and
// event parameter with the ones from the previous examples
dojo.connect(object, "fireEvent", function() {
    console.debug("dojo.connect anonymous function");
});
object.fireEvent();//dojo.connect anonymous function
</script>
<body>
    <button id="btn">Test dojo.connect</button>
</body>
</html>
```

---

If you choose a DOM-node as the obj parameter, then the events that can be fired and passed as the second parameter are the common JavaScript-events that you would be connecting to without Dojo, for example, onclick, onblur, onfocus, onkeydown, onkeypress.

## Disconnecting events

The dojo.connect function returns a connect-handle that can be used to delete a connection between a DOM-event and a function or delete a connection between two functions, as shown in Example 9-27.

*Example 9-27 Disconnecting an event*

```
var handle = dojo.connect(...);
dojo.disconnect(handle);
```

---

## Passing and accessing parameters

If a DOM-event is fired then the browser passes the event-object to the handler function. If you connect two functions, then the arguments that you pass to the source function are also passed to the handler function. Example 9-28.

*Example 9-28 Passing and accessing parameters in events*

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="isDebug:true"></script>
<script type="text/javascript">
    var object = {
        fireFunctionEvent: function(param1, param2) {},
        testFunctionEvent: function(param1, param2) {
            console.debug(param1 + ":" + param2);
        },
    },
```

```

    testDomEvent: function(event) {
        //you can access all event properties
        //following prints 'click' when you click on the button
        console.debug(event.type);}
};

// when clicked on the button, the JavaScript event object is passed
// to testDomEvent
var handle = dojo.connect(dojo.byId("btn"), "onclick", object,
    "testDomEvent");

dojo.connect(object, "fireFunctionEvent", object,
    "testFunctionEvent");
// following prints 'param1:param2'
object.fireFunctionEvent("param1", "param2");
</script>
<body>
    <button id="btn">Test dojo.connect</button>
</body>
</html>

```

---

## The event object

If you use a DOM-node as the event source, following cross-browser properties that are added to the event object by Dojo are worth mentioning:

- ▶ `event.target`: This is the DOM-element that generated the event.
- ▶ `event.preventDefault()`: This call on the event prevents the browser default behavior. For instance, if you have a onkeydown event-handler attached to a textbox with `event.preventDefault()` as the first line, then pressing a key will not cause the appropriate character to appear in the textbox.
- ▶ `event.stopPropagation()`: This call prevents the firing of an event to the parent node of the event source.

The function `dojo.stopEvent(e)` combines both the `preventDefault` and the `stopPropagation` functions, and if utilized should be placed as the first line in the event handler.

Dojo provides a map for keyboard key codes that improve the readability of the code, as shown in Example 9-29.

*Example 9-29 Handling keypress events*

---

```
dojo.connect(dojo.byId("inp"), "onkeypress", function(e) {
    if(e.keyCode == dojo.keys.ENTER) {
        console.debug("Enter was pressed!");
    }
});
```

---

All keys can be found at:

<http://api.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.keys>

## 9.4.9 Drag and drop

With Dojo, you can extend your browser application with drag and drop (DnD) functionality.

The `dojo/dnd` directory contains all the classes needed for handling DnD operations. In this section we only cover the basics of drag and drop programming with Dojo. For a more sophisticated example and enterprise business usage of DnD, check out the Plants by WebSphere sample in 9.14, “Sample: PlantsByWebSphere application” on page 438. The full Dojo DnD documentation can be found at:

[http://docs.google.com/View?docid=d764479\\_11fcs7s397](http://docs.google.com/View?docid=d764479_11fcs7s397)

To drag an element on the panel, you must mark its DOM node as a drag source. To drop an element, for instance, into a shopping cart, you first need to mark the area that you wish to drop into as the drop target. Example 9-30 shows a basic example utilizing two lists. You can simply move names from the first list to the second by dragging a name and dropping it into the second list.

*Example 9-30 Usage of Dojo’s drag and drop functionality*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="parseOnLoad: true"></script>
<script type="text/javascript">
    dojo.require("dojo.dnd.Source");
    dojo.require("dojo.parser");
</script>
<body>
    <div dojoType="dojo.dnd.Source">
```

```
<div class="dojoDndItem" dndType="mySource"><div>drag
source</div></div>
<div class="dojoDndItem" dndType="mySource"><div>drag
source</div></div>
<div class="dojoDndItem" dndType="mySource"><div>drag
source</div></div>
</div>
<br/>
<div dojoType="dojo.dnd.Target" accept="mySource">
  <div>drag target</div>
  <div>drag target</div>
</div>
</body>
</html>
```

---

All you need to do is add the dojo-proprietary attribute `dojoType` to the element and then assign either `dojo.dnd.Source` or `dojo.dnd.Target` to it. For a drag source you must add the `dndType` attribute and assign it an arbitrary string to define a type for the source. If you want to drop that type of source onto the target, add the `accept` attribute to the target element and assign it the type of the source that you want to drop.

#### 9.4.10 Handling the back-button problem

Using XMLHttpRequest for asynchronous client/server communication and dynamically updating parts of the browser window has the drawback that the back and forward buttons no longer function properly. Asynchronous changes are not added to the browser history. Furthermore, the page contents cannot be bookmarked, as the address bar does not change.



With some additional effort you can include the `dojo.back` module to add back-button functionality to your application. The main idea is to set a state object with a back and forward callback functions that are called whenever the user clicks either the back or forward button of the browser. Example 9-31 shows what the state object looks like.

*Example 9-31 State object for back-button handling in Dojo*

---

```
var state = {
  back: function() {
    // handle back button
  },
  forward: function() {
    //handle forward button
  },
  // if a string, then taken as the fragment identifier
  // if true, then Dojo generates an identifier
  changeUrl: "anIdentifier"
};
```

---

To allow for bookmarking of the current page you can use the `changeUrl` property to append a fragment identifier to the URL in the browser address bar. Setting a fragment identifier does not force the browser to reload the Web page. It can be used to create bookmarks for different states of the application. Setting `changeUrl` to `true` causes Dojo to generate an identifier each time that you set a new state. If you set it to a value that does not evaluate to false (0, null, empty string, undefined), then that value is taken. If it evaluates to false or you do not set the `changeUrl` property of the state object at all, then no fragment identifier is set.

To enable back-button functionality you must set the `preventBackButtonFix` property of the `djConfig` variable to false.

At application start you define an initial state object with `dojo.back.setInitialState(state)`. Later you can add a state object to Dojo's back-button history with `dojo.back.addToHistory(newState)`. Each `addToHistory` call causes the passed state object to be added to a Dojo internal stack. Pressing back or forward just switches between the states and calls the respective back or forward function and changes the fragment identifier of the URL if needed.

Example 9-32 gives you an idea of how you might implement back-button functionality in your application and handle bookmarks used to access the application.

*Example 9-32 Usage of Dojo's back-button functionality*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="preventBackButtonFix: false"></script>
<script type="text/javascript">
    dojo.require("dojo.back");
    dojo.require("dojo.parser");
</script>
<script type="text/javascript">
    // this is the function that restores a specific state of the
    // application
    // in this case it is populating the two textboxes
    function handleParams(param) {
        var params = decodeURIComponent(param).split(":");
        dojo.byId("inp1").value = params[0];
        dojo.byId("inp2").value = params[1];
    };

    // in this example the two strings that are typed into the both
    // textboxes are taken as the fragment identifier
    // milliseconds are appended to create a unique identifier
    function createState(param1, param2) {
        return {
            changeUrl: encodeURIComponent(param1 + ":" + param2 + ":" +
                new Date().getMilliseconds()),
            back: function(){
                handleParams(this.changeUrl);
            },
            forward: function() {
                handleParams(this.changeUrl);
            }
        }
    };

    dojo.addOnLoad(function() {
        // get the fragment identifier of the URL the user typed in if
        // exists
        var hash = window.location.hash;
        // some browsers include the hash sign, remove it
        hash = hash.charAt(0) == "#" ? hash.substring(1) : hash;
```

```

// IE does not decode the encoded identifier automatically like
// FF does
hash = decodeURIComponent(hash);
// if no fragment identifier appended, hash = ":", so
// handleParams does not break
// if the fragment identifier is appended, then the first access
// to the application is done with a bookmark and the application
// auto populates the textboxes with the strings that are passed
// in the fragment identifier
hash = (hash == "") ? ":" : hash;
handleParams(hash);
var args = hash.split(":");
//this is the initial state
dojo.back.setInitialState(createState(args[0], args[1]));
// each time the button is clicked the content of the textboxes
// are saved in the browser history
dojo.connect(dojo.byId("btn"), "onclick", function() {
    dojo.back.addToHistory(createState(dojo.byId("inp1").value,
    dojo.byId("inp2").value));
});
});
</script>
<body>
<!-- this hack is needed for IE to initialize the undo stack -->
<script>
    dojo.back.init();
</script>
    Input1: <input id="inp1" />
    Input2: <input id="inp2" />
    <button id="btn">Add to browser history</button>
</body>
</html>

```

---

The application presents two text input elements and a button. You can enter values into the boxes. Pressing the button adds the values to the history and you can use the back and forward button to undo or redo changes that you made to the textboxes. You can use bookmarks to jump to a specific state of the application. For instance, you can append the fragment identifier #Hi:there! and the first textbox is populated with *Hi* and the second with *there!*.

Once you press the button to save the entries into the history, the fragment identifier in the address bar changes and you can copy that link as a bookmark. When you open another browser, simply insert the bookmark into the address bar and the application will jump into the previously bookmarked state.

The application was successfully tested with Firefox 2.0+ and IE6. However, the dojo.back module does not always work properly for all browsers. Notes on browser compatibility can be found in the dojo/back.js file or at:

<http://dojotoolkit.org/book/dojo-book-0-9/part-3-programmatic-dijit-and-dojo/back-button/browser-compatibility>

### 9.4.11 Selecting DOM nodes with Dojo's query engine

Dojo provides a powerful, high-performance query engine that can be used to search for DOM nodes with specific CSS-selectors. You can find a performance evaluation of the dojo.query function at:

<http://dojotoolkit.org/node/336>

#### Querying DOM elements

Example 9-33 provides a sample of the usage of the dojo.query function. A rich set of CSS3 selectors is supported. The list of supported selector types can be found at:

<http://api-staging.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.query>

List of CSS3 selectors at:

<http://www.w3.org/TR/css3-selectors>

*Example 9-33 Querying DOM elements with dojo.query*

---

```
// all nodes with a class 'aClass'
dojo.query(".aClass")
// all nodes with the id 'anId'
dojo.query("#anId")
// all nodes with a div tag
dojo.query("div")
// all div nodes with a class 'aClass'
dojo.query("div.aClass")
// all nodes whose 'attribute' attribute value is 'anAttribute'
dojo.query("[attribute=anAttribute]")
// all nodes with an attribute named 'attribute'
dojo.query("[attribute]")
// the first div node found
dojo.query("div:first-child")
// the second div node found
dojo.query("div:nth-child(2)")
// a div node with the id 'anId'
dojo.query("div#anId")
```

```
// a span node that is child of a div node
dojo.query("div > span")
// a div node whose 'attribute' attribute value contains 'anAttr'
dojo.query("div[attribute*=anAttr]")
```

---

The result type of `dojo.query` is a JavaScript array containing the nodes found.

### Extended usage functions

Dojo provides useful functions that can be applied to the result array of the `dojo.query` call. Example 9-34 shows some of the more interesting functions that you can apply to the result array. A complete description of all the functions that may be used with `dojo.query` results can be found at:

<http://api-staging.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.NodeList>

*Example 9-34 Extended usage of `dojo.query`*

---

```
//set background color of all div nodes
dojo.query("div").style("backgroundColor", "yellow");
//fade out all div nodes, dojo.require("dojo.NodeList-fx") needed to
// use all effect functions
dojo.query("div").fadeOut({duration: 2000}).play();
dojo.query("div").forEach(function(element, index) {
    //do something
});
//add class to all div nodes
dojo.query("div").addClass("addClass");
//can also apply other DOM events
dojo.query("aLink").onclick(function(event){
    //do something
});
//set background color of every second div node
dojo.query("div").filter(function(element, index) {
    return index % 2;
}).style("backgroundColor", "yellow");
```

---

## 9.5 Dojo's pub/sub event-system

When creating a loosely coupled application using Dojo widgets, it is important that these components are able to communicate with each other without having to implement hardwired configuration or protocols. Dojo has a publish subscribe mechanism built into the core module that supports simple APIs and requires minimal configuration of the Dojo library to use.

The pub/sub event-system allows any components loaded within a single HTML page to communicate with each other. This may be a portal system or between a controller and its view components. This is a common pattern when you have numerous calls being made asynchronously back to the server and you want to manage how the views are updated.

## **Name space**

When designing a publish subscribe system you need to think about the name spaces that you will use. These will be used throughout a page and consideration should be taken to avoid conflicts and duplication. Try and use a hierarchy of name elements separated with underscores or dots. This will allow you to understand the scope of the published data. Dojo does not support sophisticated *topic structs* in which wild cards can be used.

## **dojo.publish**

The `dojo.publish` method enables a component, widget, or JavaScript method to broadcast data to a topic. If there has been a corresponding `dojo.subscribe` issued to the same topic then data on that topic will be published to the subscriber.

The `publish` method takes two arguments. The first being the name of the topic to which to publish and the second being an array of data elements. Example 9-35 shows the `publish` function.

*Example 9-35 Example dojo.publish*

---

```
var hello = "Hello Everybody out there";
dojo.publish("helloworld", [hello] );

var myarray = ["One", "Two", "Three"];
dojo.publish("numbers", myarray);
```

---

You can see from this example that it is a simple API. You just call the `publish` method with the topic and data.

## **dojo.subscribe**

The `dojo.publish` command is not very useful if no one is listening for the data being published. To enable a decoupled handler to receive publish events you must subscribe to a topic being published to.

To perform a subscribe you must specify a JavaScript function to be invoked when the topic receives notification of a publish. You can do this inline with the `dojo.subscribe` call or specify one at the `dojo` component level or with a JavaScript function.

Once you have defined the JavaScripts method you need to place a subscription call in your logic. It makes the most sense to do this during the initialization phase of your widget component before any publish events are triggered. Example 9-36 shows the subscribe function.

*Example 9-36 Example dojo.subscribe*

---

```
function myTopicHandler(topicData)
{
    alert(topicData);
}

dojo.subscribe("helloworld",null,"myTopicHandler");
```

---

If you issue `dojo.subscribe` within a Dojo object or widget you set the null parameter to *this* in order to reference the object instance itself. You can see that this simple API is very powerful in a componentized development environment where you want to exchange information between loosely coupled components.

### **dojo.unsubscribe**

The final method to complete the set is the `dojo.unsubscribe` method, which is used to remove the subscription to a topic. You may find that you are dynamically creating subscriptions and may eventually want to unsubscribe from them or you want to clean up your subscriptions as your widget or component is destroyed or closed. Example 9-37 shows the unsubscribe function.

*Example 9-37 Example dojo.unsubscribe*

---

```
//dojo.subscribe returns a handle object
var handle = dojo.subscribe("helloworld", null, "myTopicHandler");
//simply pass the handle to dojo.unsubscribe to remove the subscription
dojo.unsubscribe(handle);
```

---

By threading the Dojo toolkit pub/sub mechanism into your applications you create powerful, reusable components that are decoupled from the data and events. This enables Model View Controller design patterns to be used within Dynamic Ajax applications.

## 9.6 XHR, IFrames, JSONP, and RPC

This section describes the Dojo functions that enable a client-server connection using XMLHttpRequest (XHR), JSONP, IFrames, and Remote Procedure Calls (RPC).

### 9.6.1 XMLHttpRequest

Dojo provides a cross-browser abstraction of the XMLHttpRequest object used for synchronous or asynchronous communication between the client and the server. With Dojo you no longer need to concern yourself with the different browser implementations of that object.

Dojo provides the four main functions:

- ▶ `dojo.xhrGet(args)` for GET-requests
- ▶ `dojo.xhrPost(args)` for POST-requests
- ▶ `dojo.xhrPut(args)` for PUT-requests
- ▶ `dojo.xhrDelete(args)` for DELETE-requests.

#### Parameters for `dojo.xhr` functions

The argument `args` is an object with key-value mappings that can contain the following properties:

- ▶ `url (String)`: The path name of the requested resource on the server. You can use any server-side technology that can receive a request and send back a response to the client such as a Java Servlet, Java Server Page, PHP, Perl CGI, ASP, and so on. Furthermore, the URL key can be used to read out static files on the server like normal text files.
- ▶ `content (Object)`: Content contains the properties that are passed as arguments to the request if it is passed a JSON-object in the format `{key1:value1,key2:value2}`. Dojo serializes the object to `key1=value1&key2=value2` and appends it to the request URL for a GET. For a POST, the parameters are added to the request body.
- ▶ `timeout (Integer)`: The time to wait for the response in milliseconds. If the specified time passes, the error/handle callback is invoked.
- ▶ `form (DOM node)`: Pass the node of a form on your Web page or the ID of the form and Dojo then extracts the values and sends them to the server.
- ▶ `preventCache (Boolean)`: Some browsers like IE6 or IE7 cache the response content for GET requests. This means that if a request is the same as a previous request, then the browser does not send that request to the server and instead passes the cached response back to the caller. In certain situations this behavior can produce undesirable results. To prevent the



browser from caching the response set `preventCache` to `true`. If not set, the default value is `false`.

- ▶ `handleAs` (String): Dojo can parse the server response and pass the resulting data object to the callback function. The supported formats are:
  - `text` (default): The response from the server is passed to the callback as normal text.
  - `json`: The server response is parsed and the resulting JSON object is passed to the callback.
  - `json-comment-optional`: One technique to prevent JavaScript hijacking is to enclose the server response in comment characters, for example, `/*[{key:'val'}]*/`. If `handleAs` is set to `json-comment-optional`, then the response may be enclosed with comment characters, which are later removed before the response is evaluated.
  - `json-comment-filtered`: Similar to `json-comment-optional`, but stricter. The response must be enclosed in comment characters. Otherwise, Dojo does not evaluate the response and generates the error `JSON was not comment filtered`.
  - `javascript`: The response is evaluated as JavaScript.
  - `xml`: The response is parsed as XML and the object is passed to the callback.
- ▶ `load` (Function): The callback function for a successful request. The callback is passed two arguments in the format `function(response, ioArgs)`:
  - `response`: The server response in the format specified for the `handleAs` property.
  - `ioArgs`: An object that contains information about the XHR call. It has the following properties:
    - `args`: The object that is passed to the `dojo.xhr` function.
    - `xhr`: The `xhr` object that is used for the request. It also contains the properties `responseText`, `responseXML`, `status`, and `statusText`. If the response is valid XML, then `responseText` is the response as a string and `responseXML` a reference to the parsed XML document. If the response is not XML, then only the `responseText` is populated and `responseXML` is null. `status` is the HTTP status code and `statusText` the HTTP status text. A full list of HTTP status and texts can be found at: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>
    - `url`: The URL that is used for the request.

- query: The parameters that are passed in the request body. Only populated for POST and PUT.
- handleAs: The data format used for parsing the server response.
- ▶ error (Function): The callback function for an unsuccessful request. The callback is passed two arguments in the format function(error, ioArgs):
  - error: An error object that contains the error message, for example, Error: bad http response code: 403. If an xhr-call was canceled or timed out, then additionally the error.dojoType is set to cancel or timeout.
  - ioArgs: An object that contains information about the XHR call. It has the following properties:
    - args: The object that is passed to the dojo.xhr function.
    - xhr: The xhr object that is used for the request. It also contains the properties responseText, responseXML, status, and statusText. If the response is valid XML, then responseText is the response as a string and responseXML a reference to the parsed XML document. If the response is not XML, then only the responseText is populated and responseXML is null. status is the HTTP status code and statusText the HTTP status text. A full list of HTTP status and texts can be found at: <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>
    - url: The URL that is used for the request.
    - query: The parameters that are passed in the request body. Only populated for POST and PUT.
    - handleAs: The data format used for parsing the server response.
- ▶ handle (Function): The handle callback will be invoked for both a successful and an unsuccessful request. Use this callback if you do not want to use either the load or the error method. This callback is passed two arguments in the format function(data, ioArgs):
  - data: If the request is successful, it contains the response in the format specified for handleAs. If the request was unsuccessful it contains the error message.
  - ioArgs: An object that contains information about the XHR call. It has the following properties:
    - args: The object that is passed to the dojo.xhr function.
    - xhr: The xhr object that is used for the request. It also contains the properties responseText, responseXML, status, and statusText. If the response is valid XML, then responseText is the response as a string and responseXML a reference to the parsed XML document. If the response is not XML, then only the responseText is populated and

responseXML is null. status is the HTTP status code and statusText the HTTP status text. A full list of HTTP status and texts can be found at:

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

- url: The URL that is used for the request.
- query: The parameters that are passed in the request body. Only populated for POST and PUT.
- handleAs: The data format used for parsing the server response.
- ▶ synch (Boolean): Set this to true if you want the request to block. The default setting is false, which allows asynchronous communication with the server.
- ▶ headers (Object): Use headers if you want to send additional HTTP header information. The format of the value is an object with key-value mappings. A list of all possible header definitions can be found at:  
<http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>
- ▶ contentType: Sets the content type if you do not want to use the headers property for setting it.

## xhrGet

Example 9-38 shows how to use the properties previously described with the xhrGet.

*Example 9-38 Usage of dojo.xhrGet*

---

```
dojo.provide("ibmDijit.Test");
dojo.declare("ibmDijit.Test", null, {
  doGet: function(){
    var deferred = dojo.xhrGet ({
      url: "/xhr/JsonServlet",
      handleAs: "json",
      synch: false,
      load: this.handleResponse,
      error: dojo.hitch(this, function(error, ioArgs) {
        if(error.dojoType == "cancel") {
          console.debug("XHR cancelled!");
        } else if(error.dojoType == "timeout") {
          console.debug("XHR timed out!");
        } else {
          console.debug("Request error! Code: "
            + ioArgs.xhr.status
            + ", Text: " + ioArgs.xhr.statusText);
        }
      })
    },
    return error;
  })
});
```

```

        timeout: 5000,
        content: {"forename": "Mehmet", "surname" : "Akin"}
    });

    },

    handleResponse: function(response) {
        console.debug("Successful request!");
        // do something with the response
        return response;
    }
});

```

---

Note that in order to have access to the `this` reference inside the callback context you have to use `dojo.hitch`. It is good practice to always return from a callback method. If you do not return from the callback handler for a successful request and an error occurs while handling the response in that function, then the error callback is also called and passed the JavaScript error.

Example 9-39 shows how to use the `handle` callback if you do not want to use `load` and `error`.

---

*Example 9-39 Handling responses with the `handle` property*

---

```

handle: dojo.hitch(this, function(data, ioArgs) {
    if(data instanceof Error) {
        // handle error params
    } else {
        // handle success params
    }
})

```

---

## **xhrPost**

The `dojo.xhrGet` function automatically appends the `content` property as key-value pairs to the `URL` property.

When using `dojo.xhrPost`, the `content` property is added to the request body as key-value pairs. If your `URL` property contains parameters, they are sent as part of the `URL` request and not in the `POST` body.

Use `dojo.xhrPost` if you want to send large amounts of data or prefer sending the parameters within the request body.

Example 9-40 shows how you can easily transfer all of the form parameters to the server.

*Example 9-40 Posting a form with dojo.xhrPost*

---

```
<html>
<head>
<script type="text/javascript" src="dojo/dojo.js"></script>
<script type="text/javascript">
    function sendForm() {
        dojo.xhrPost({
            url: "/xhr1/JsonServlet",
            handleAs: "json",
            load: function(response) {
                // handle response
                return response;
            },
            error: function(error, ioArgs){
                // handle the error
                return error;
            },
            form: "myForm",
            // some additional params
            content: {"key1": "val1", "key2" : "val2"}
        });
    }
</script>
</head>
<body>
    <form id="theForm">
        First name: <input type="text" name="firstname">
        <br>
        Last name: <input type="text" name="lastname">
        <br>
        Profession: <input type="text" name="lastname">
        <br>
        <input type="button" onclick="sendForm()" value="Send via
        xhrPost!">
    </form>
</body>
</html>
```

---

## xhrPut and xhrDelete

You can use `dojo.xhrPut` for PUT-requests and `dojo.xhrDelete` for DELETE-requests in the same way as you do for `dojo.xhrGet` and `dojo.xhrPost`.

## Adding multiple callbacks to the request

The `dojo.xhr` functions return a `dojo.Deferred` object that allows for dynamically adding new callbacks or canceling a request. For a detailed description of the `dojo.Deferred` object, check the API description at:

<http://redesign.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.Deferred>

Example 9-41 gives an overview of how callbacks can be added dynamically and how a request can be cancelled. Callbacks are called in the order of connection to the `dojo.Deferred` object.

*Example 9-41 Adding multiple callbacks to an xhr object*

---

```
var deferred = dojo.xhrPost({
    url: "/xhr/JsonServlet",
    handleAs: "json",
    content: {"forename": "Mehmet", "surname" : "Akin"}
});

// add a callback for successful request
deferred.addCallback(function(response) { /*...*/ });

// addCallback returns dojo.Deferred object, possible to add several
// callback at once
deferred.addCallback(function(response) { /*...*/ })
    .addCallback(function(response) { /*...*/ });

// add an error callback
deferred.addErrback(function(error) { alert(error.dojoType) });

// callback for both an error and successful request
deferred.addBoth(function(data) { alert("both") });

// cancel the request -> error callback called with
// error.dojoType="cancel"
deferred.cancel();
```

---

## 9.6.2 JSONP cross-domain scripting with Dojo

JSONP is a technique to load data from servers in other domains. One solution is that the client defines a function that handles JSON and appends the function name to the src property of a script tag, for example, `<script type="text/javascript" src="http://server/servlet?param=MehmetAkin&jsonp=callback"></script>`. The server must be able to handle this additional parameter. If the server is JSONP enabled, it encloses the JSON response with the callback function name, for example, `callback(" + JSON string + ")`, and the callback function is invoked once the response arrives. More information about JSONP can be found at:

<http://bob.pythonmac.org/archives/2005/12/05/remote-json-jjsonp>

Many Web sites such as Yahoo!, Google, Twitter, Flickr, and AOL already support JSONP-style Web services for accessing JSON data.

Dojo provides the module `dojo.io.script` to realize JSONP calls in an easy way. Example 9-42 shows the usage of the `dojo.io.script` module together with the JSONP Web service API for the Yahoo! search engine.

*Example 9-42 Cross-domain scripting with dojo.io.script*

---

```
<html>
<head>
<style type="text/css">
    #resultContainer {
        width: 35%;
        font-family: arial,helvetica,clear,sans-serif;
    }
    .result {padding-bottom: 10px;}
    .title {color: #0000DE; font-size: 1em;}
    .summary {font-size: 0.8em;}
    .displayUrl {color: #008000; font-size: 0.8em;}
</style>
<script type="text/javascript" src="dojo/dojo.js"></script>
<script type="text/javascript">
    dojo.require("dojo.io.script");
</script>
<script type="text/javascript">
    var mySearch = "websphere feature pack Web 2.0";
    dojo.addOnLoad(function() {
        var resultContainer = dojo.byId("resultContainer")
        dojo.io.script.get({
            url:"http://search.yahooapis.com/WebSearchService/V1/
                + WebSearch",
```

```

callbackParamName: "callback",
content: {query: mySearch, appid: "YahooDemo", output:
    "json"},
load: function(response, ioArgs) {
    dojo.forEach(response.ResultSet.Result, function(el) {
        var result = dojo.doc.createElement("div");
        var tmpDiv = dojo.doc.createElement("div");
        var link = dojo.doc.createElement("a");

        // add title
        tmpDiv.appendChild(link);
        link.innerHTML = el.Title;
        link.href = el.Url;
        dojo.addClass(link, "title");
        result.appendChild(tmpDiv);
        // add summary
        tmpDiv = dojo.doc.createElement("div");
        tmpDiv.innerHTML = el.Summary;
        dojo.addClass(tmpDiv, "summary");
        result.appendChild(tmpDiv);
        // add displayed url
        tmpDiv = dojo.doc.createElement("div");
        tmpDiv.innerHTML = el.DisplayUrl;
        dojo.addClass(tmpDiv, "displayUrl");
        result.appendChild(tmpDiv);
        // add result to body
        dojo.addClass(result, "result");
        resultContainer.appendChild(result);
    });
    return response;
},
error: function(error, ioArgs) {
    //handle error
    return error;
}
});
})
</script>
</head>
<body>
    <div id="resultContainer"></div>
</body>
</html>

```

---



You can use `dojo.io.script.get` similar to the `dojo.xhr` functions. You can apply the same properties, except for the `handleAs` property, since it is implied by the `callbackParamName`. (The response is a JSONP call returning JSON.) The `callbackParamName` is the parameter key that contains the callback function name. For instance, Yahoo! expects the parameter name to be `callback` in order to send back JSONP-formatted data. Dojo automatically passes its own function name to the callback parameter. When the response arrives, it is then mapped to the `load`, `error`, or `handle` function. Documentation on how to use the Yahoo! API can be found at:

<http://developer.yahoo.com/search/Web>

Note that JSONP security issues are not covered in this book.

Example 9-42 on page 323 handles the Yahoo! JSON search results and displays them, along with some styling. Just change `content.query` and try it yourself. With just a few lines of code you can create JavaScript that enables you to integrate the Yahoo! search engine into your Web page.

The results displayed by this script are shown in Figure 9-4. It appears similar to the actual search results displayed on the Yahoo! Web site.

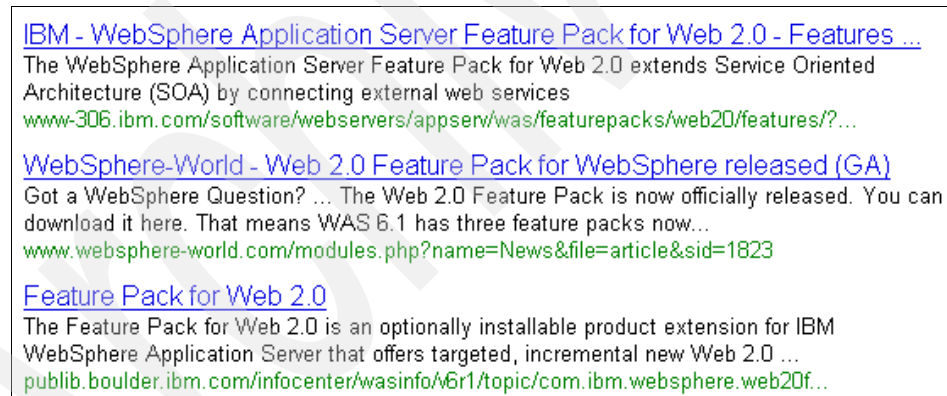


Figure 9-4 Results of Yahoo! JSONP search

### 9.6.3 Uploading files with IFrames

Since it is not possible to use the `XMLHttpRequest` object for forms with file uploads, you cannot use `dojo.xhr` functions to perform that task. IFrames, however, do support file uploads. Dojo provides the module `dojo.io.iframe` to make an IFrame request. You can use `dojo.io.iframe.send` in a manner similar to the `dojo.xhr` functions. The default transport type is `POST`. If you want to change that, a property called `method` with the value `GET` must be set. The property

handleAs can only be text/plain, text/html, text/javascript, text/json, or application/json. Example 9-43 shows how you can upload a file to a server.

*Example 9-43 Uploading files with IFrames*

---

```
<html>
<head>
  <script type="text/javascript" src="dojo/dojo.js"></script>
  <script type="text/javascript">
    dojo.require("dojo.io.iframe");
  </script>
  <script type="text/javascript">
    function uploadFile(){
      dojo.io.iframe.send({
        form: dojo.byId("fileUpload"),
        handleAs: "application/json",
        handle: function(data, ioArgs) {
          if(data instanceof Error) {
            // handle error
          } else {
            // successful upload
          }
        }
      });
    }
  </script>
</head>
<body>
  <form action="/xhr/UploadServlet" id="fileUpload"
    method="POST" enctype="multipart/form-data">
    <input type="file" name="urFile">
    <input type="button" onclick="uploadFile(); return false;"
value="Upload file!">
  </form>
</body>
</html>
```

---

To detect when the file upload has completed, the only cross-browser solution is to send an HTML document with a text area that contains the response to the client, for example,

```
<html><head></head><body><textarea>{'key':'value'}</textarea></body></html>
>. For the handleAs parameter text/html it is enough to return a normal HTML document.
```

## 9.6.4 Remote Procedure Calls

Dojo provides the module `dojo.rpc.RpcService` to implement remote procedure calls.

The first thing needed is a Simple Method Description (SMD) to define the RPC functions. The SMD is in JSON format and can be defined either in a separate file as a JSON string or a JavaScript object containing JSON data.

Example 9-44 shows the structure of an SMD.

*Example 9-44 Sample of a simple method description*

---

```
{
  "serviceType": "JSON-RPC",
  "serviceURL": "/rpc/RpcServlet",
  "methods": [
    {
      "name": "addUser",
      "parameters": [
        { "name": "forename" },
        { "name": "surname" }
      ]
    },
    {
      "name": "deleteUser",
      "parameters": [
        { "name": "forename" },
        { "name": "surname" }
      ]
    }
  ]
};
```

---

The SMD above defines the two functions `addUser(forename, surname)` and `deleteUser(forename, surname)`. The request is sent to the `serviceURL`.

Example 9-45 shows how to use the `dojo.io.JsonService`, a subclass of `dojo.io.RpcService`, to call a remote procedure.

*Example 9-45 Implementing a remote procedure call*

---

```
// pass path to yourSMD.smd file
var rpc = new dojo.rpc.JsonService("/pathToSMD");
// or pass SMD as JSON object
var rpc = new dojo.rpc.JsonService(JSONObject);
```

```
// or pass SMD as JSON string
var rpc = new dojo.rpc.JsonService({smdStr: "JSONstring"});

// call the remote method
var deferred = rpc.addUser("Niyazi","Akin");
deferred.addCallback(function(response) {
    // callbackMethod
});
rpc.addUser("Ihsan","Akin").addCallback(function(response) {
    // callbackMethod
});

deferred = rpc.deleteUser("Nuriye","Akin");
deferred.addErrback(function(response) {
    // errorCallbackMethod
});
rpc.deleteUser("Emine","Akin").addErrback(function(response) {
    // callbackMethod
});
```

---

Dojo sends a JSON string containing all of the parameters to the server, for example:

```
{"params": ["Ihsan", "Akin"], "method": "addUser", "id": 2}
```

For an RPC object an id property is incremented each time a method within that object is called. It is the current number of total remote procedure calls.

## 9.7 The Dojo widget library

Dijit is Dojo's widget system layered on top of the core modules. This section describes the widget modules. It explains how to use and customize the look and feel of widgets provided by Dijit. How to build your own customized widgets from scratch or extend existing ones is discussed in 9.8, "Creating custom Dojo widgets" on page 350.

### 9.7.1 Widget basics

A widget is constructed from JavaScript. Optional parts are an HTML template, CSS definitions, and images:

- ▶ JavaScript (\*.js): A JavaScript class that contains the logic of the widget. It can access the elements of the HTML template if the widget uses one.
- ▶ HTML template (\*.htm, \*.html): A widget can have a graphical representation with an HTML template or can be used without a template to extend passed HTML elements with functionality. If the HTML template is simple, it can be used as a string in the JavaScript file.
- ▶ CSS definitions (\*.css): A widget can have visual style definitions that are used by the HTML template or the JavaScript file.
- ▶ images (\*.jpg, \*.gif, \*.png, and so on): Images can be used by the JavaScript file in the HTML template or CSS style definitions.

Imagine a widget as an HTML element that is already extended with functionality by Dojo, for instance, a normal table enhanced by a sorting function or a textbox that supports input validation.

### 9.7.2 Overview of existing widgets

This section provides an overview of the widgets that are available with the dijit module. The concepts of how to use widgets, both in a declarative and a programmatic way, are also presented.

For a declarative or programmatic usage for all of the widgets, check the dijit/tests directory, the Dojo API at <http://dojotoolkit.org/api>, or the Dojo Toolkit Web site.

## Buttons

If you want a browser and operating system (OS) independent-looking button, use the button widget. Use the ComboBox or DropDownButton if you need to provide other options in a pop-up. Compared to the ComboBox, a select the DropDownButton always opens the options pop-up, whereas the ComboBox can be used as a button itself without opening the pop-up.

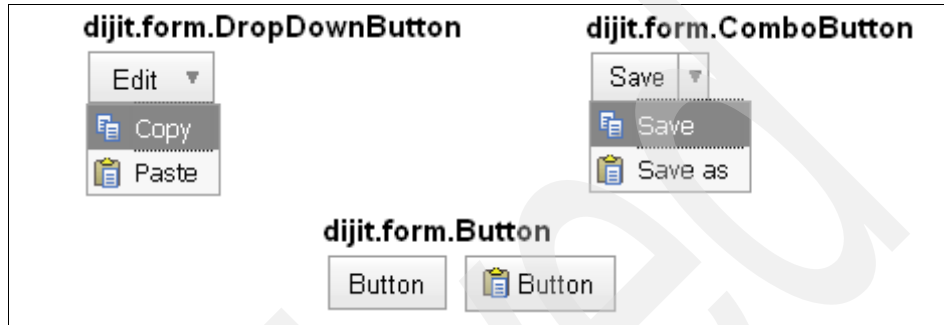


Figure 9-5 Button widgets

## Menu and toolbar

You can use the menu widget with or without other widgets, for example, when opening the menu after a select a DropDownButton or right-click the window.

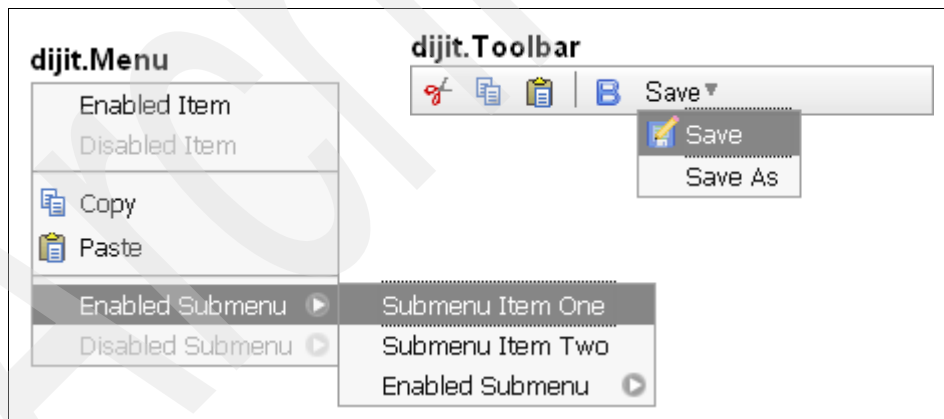


Figure 9-6 Menu and toolbar widgets

## Text editing

Use the editor widget to add rich text editing capabilities to your application. Additional text editing widgets are:

- ▶ The `dijit.form.TextArea` widget
- ▶ A `<textarea>` tag extension that dynamically increases the height of the textbox when typing
- ▶ A `dijit.InlineEditBox`

The `InlineEditBox` turns static text into a textbox and back again if the user wants to change the text dynamically. See Figure 9-7.



Figure 9-7 Text editing widgets

## Input assist and validation

You can create your own type of validation widgets by using the `dijit.form.ValidationTextBox` or extending the `dijit.form.TextBox` widget (Figure 9-8).

**dijit.form.DateTextBox**  
5/21/08a [!]

\* The value entered is not valid.

May

S	M	T	W	T	F	S
27	28	29	30	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31
1	2	3	4	5	6	7

2007 2008 2009

**dijit.form.TimeTextBox**  
2:00 PM

12:00 PM

1:00 PM

2:00 PM

3:00 PM

4:00 PM

**dijit.form.FilteringSelect**  
ibm red [v]

IBM Redbook WAS Feature Pack 2.0  
IBM Redbook Websphere  
IBM Redbook Residency

**dijit.form.NumberTextBox**  
string [!]

\* The value entered is not valid.

**dijit.form.NumberSpinner**  
max=50  
100 [!]

\* This value is out of range.

**dijit.form.CurrencyTextBox**  
US\$11,235.5 [!]

\* The value entered is not valid.

Figure 9-8 Widgets with input assist validation



## ColorPalette

Figure 9-9 shows the ColorPalette.



Figure 9-9 ColorPalette widget to choose colors

## CheckBox and RadioButton

Figure 9-10 shows the CheckBox and RadioButton



Figure 9-10 CheckBox and RadioButton widgets

## Slider and ProgressBar

Figure 9-11 shows the Slider and ProgressBar.

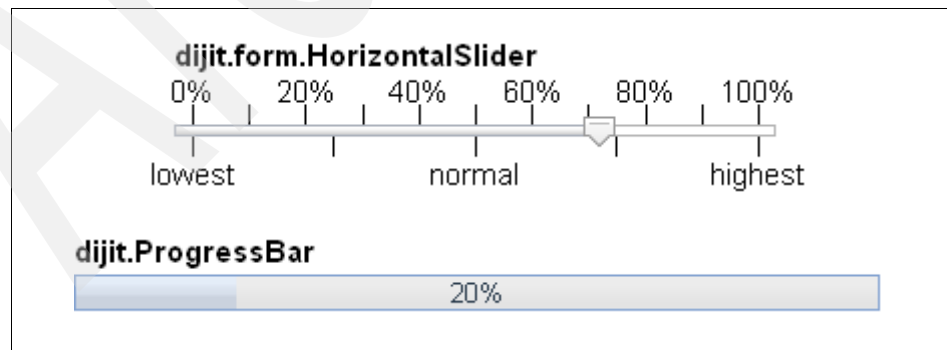


Figure 9-11 Slider and ProgressBar widget

## TitlePane and dialog

Figure 9-12 shows the TitlePane and dialog.

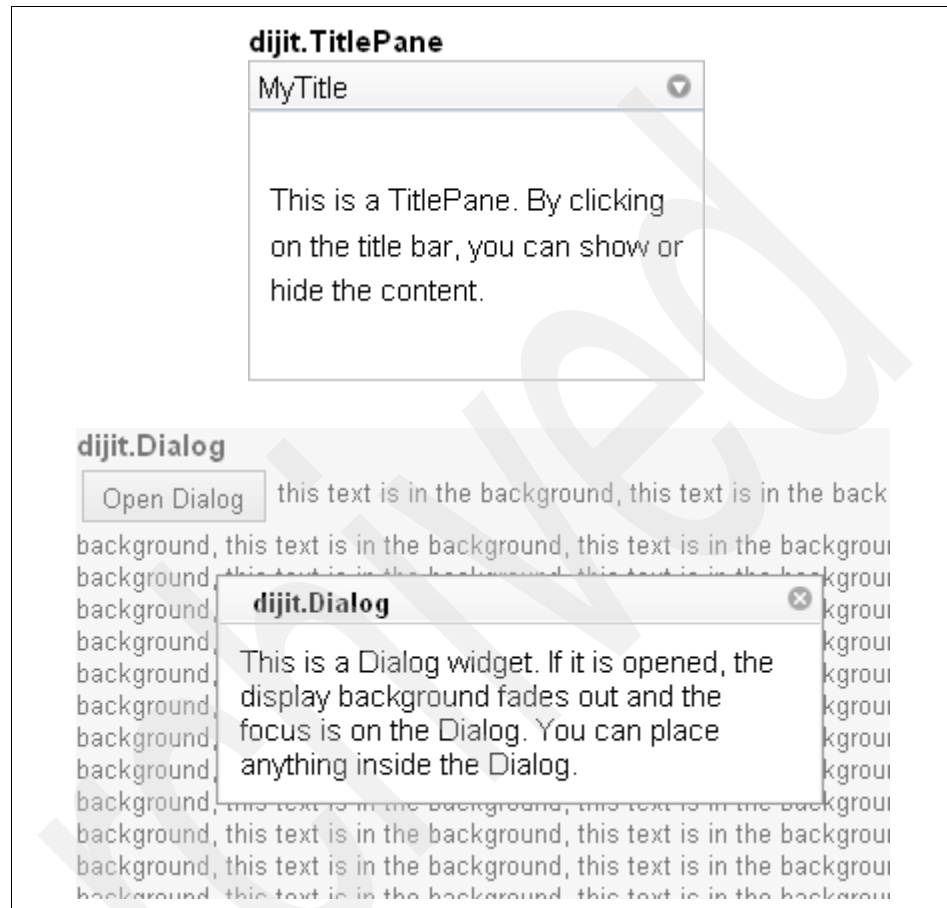


Figure 9-12 TitlePane and dialog widgets

## Tooltip and TooltipDialog

Figure 9-13 shows the tooltip and TooltipDialog widgets.



Figure 9-13 Tooltip and TooltipDialog widget

## Tree

Figure 9-14 shows the tree widget.

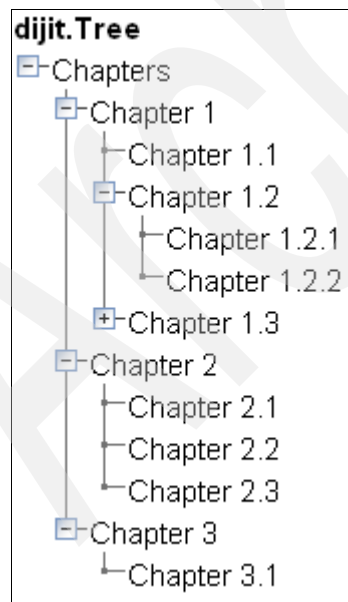


Figure 9-14 Tree widget

## Layout

Figure 9-15 shows the layout.

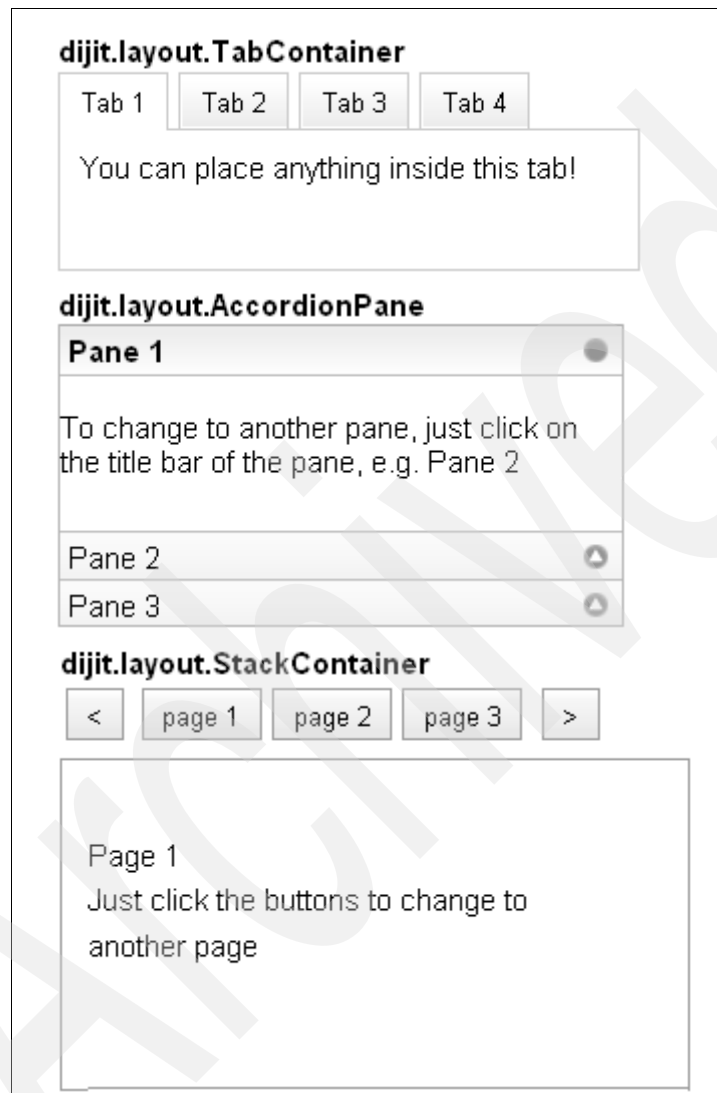


Figure 9-15 *TabContainer, AccordionPane, and StackContainer widget*

`dijit.layout.ContentPane` serves as a baseclass for other widgets. It can also be used as a standalone box for representing any type of data. In most cases you will use this widget inside of other layout widgets, for example, as a tab in a `TabContainer`.

The `dijit.layout.SplitContainer` puts a splitter bar between two `ContentPane` widgets and supports the dynamic resizing of the region within a pane. `dijit.layout.LayoutContainer` is used to display `ContentPane` widgets in a specific layout, for example, left, right, top, bottom, center.

Dojo 1.1 deprecated the `SplitContainer` and `LayoutContainer` and replaced them with `dijit.form.BorderContainer`. This is shown in Figure 9-16.

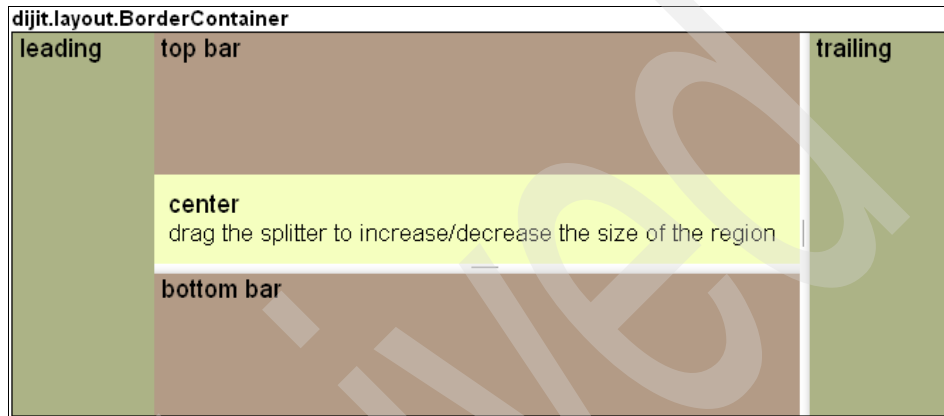


Figure 9-16 *BorderContainer widget*

### 9.7.3 Using widgets declaratively

To include a widget declaratively in your HTML markup, you need to add an HTML-tag with at least the `dojo-proprietary` attribute `dojoType` to your HTML-code, load the widget class with `dojo.require`, and set `djConfig.parseOnLoad` to `true` or parse the widget manually. Parsing is described in 9.8.5, “Using the Dojo parser” on page 361. Example 9-46 shows how to use the button widget.

Example 9-46 *Using a widget declaratively*

```
<html>
<head>
<style type="text/css">
  @import "../dijit/themes/tundra/tundra.css";
</style>
<script type="text/javascript" src="../dojo/dojo.js"
  djConfig="parseOnLoad: true, isDebug: true">
</script>
<script type="text/javascript">
  dojo.require("dijit.form.Button")
</script>
```

```
</head>
<body class=tundra>
  <div dojoType="dijit.form.Button" label="Dojo Button"></div>
</body>
</html>
```

---

The type of HTML tag needed to include a widget can differ depending on the widget type. For most widgets a simple <div> tag is enough.

The `dojoType` attribute is set to the widget type that you want to use. In this example it is a simple Dojo button. The `label` attribute is an argument that is passed to the button widget when it is created. It is the button name to be displayed. Depending on the widget type it may be necessary to pass additional arguments.

At this stage, do not worry about the import of the `tundra.css` and setting the class of the body to `tundra`. That is only required for widget styling and is discussed in 9.7.7, “Custom widget styling” on page 343.

Other widgets like the `TabContainer` expect you to pass other widgets to create it. Example 9-47 shows how to create a `TabContainer` with three tabs.

---

*Example 9-47 Using the TabContainer widget*

---

```
<html>
<head>
<style type="text/css">
  @import "../dijit/themes/tundra/tundra.css";
</style>
<script type="text/javascript" src="../dojo/dojo.js"
  djConfig="parseOnLoad: true, isDebug: true">
</script>
<script type="text/javascript">
  dojo.require("dijit.layout.TabContainer");
  dojo.require("dijit.layout.ContentPane");
</script>
</head>
<body class=tundra>
  <div dojoType="dijit.layout.TabContainer">
    <div dojoType="dijit.layout.ContentPane"
      title="Tab1">Content</div>
    <div dojoType="dijit.layout.ContentPane"
      title="Tab2">Content</div>
```

```

        <div dojoType="dijit.layout.ContentPane"
            title="Tab3">Content</div>
    </div>
</body>
</html>

```

---

Since you need Dojo-proprietary attributes to use widgets, Dojo provides its own parser to build the DOM tree out of the HTML markup, including the widgets. At this stage, you only need to know that the parser runs through the entire HTML-document and searches for tags with Dojo proprietary attributes and then replaces those tags with the appropriate Dojo code. For instance, Example 9-48 shows the markup that replaces `<div dojoType="dijit.form.Button" label="Dojo Button"></div>`.

*Example 9-48 Dojo widget markup replacement*

---

```

<div class="dijit dijitLeft dijitInline dijitButton dijitButton"
    dojoattachevent="onclick:_onButtonClick,onmouseenter:_onMouse,onmouseleave:_onMouse,onmousedown:_onMouse" widgetid="dijit_form_Button_0">
    <div class="dijitRight">
        <button id="dijit_form_Button_0" class="dijitStretch
            dijitButtonNode dijitButtonContents"
            waistate="labelledby-dijit_form_Button_0_label" wairole="button"
            type="button" dojoattachpoint="focusNode,titleNode"
            role="wairole:button" aaa:labelledby="dijit_form_Button_0_label"
            tabindex="0" aaa:valuenow="" aaa:disabled="false">
            <span class="dijitInline" dojoattachpoint="iconNode">
                <span class="dijitToggleButtonIconChar"></span>
            </span>
            <span id="dijit_form_Button_0_label" class="dijitButtonText"
                dojoattachpoint="containerNode">Dojo Button</span>
        </button>
    </div>
</div>

```

---

## 9.7.4 Widget HTML attributes

The following list describes the attributes that are set in Example 9-48 on page 339.

- ▶ **class:** These are the Dojo-generated classes needed to apply Dojo-specific styling. If you declare the class attribute when creating the widget (for example, `<div dojoType="widgetType" class="myClass1 myClass2">`), then those classes are added to the top-level node of the widget HTML.
- ▶ **dojoattachevent:** Used for connecting events via markup. This attribute is internally used by the widgets and described in 9.8, “Creating custom Dojo widgets” on page 350.
- ▶ **id, widgetid:** If you pass an ID attribute (for example, `<div dojoType="widgetType" id="myId"></div>`), then that ID is used to reference the DOM node, the top-level node of the widget, and the widget object itself. You can get the widget DOM node with `dojo.byId` and a reference to the widget object with `dijit.byId`. These functions are explained in 9.4.4, “Useful Dojo core functions” on page 285.

The attribute `widgetid` is internally used by Dojo to store references to created widgets. If no ID attribute is passed Dojo generates one internally.

- ▶ **waistate, wairole, tabindex:** These attributes are used for widget accessibility and are described in 12.6.4, “Assistive technology support” on page 602.
- ▶ **dojoattachpoint:** This attribute is internally used by widgets to reference sub-nodes inside the DOM node of the widget. The DOM nodes `domNode` and `containerNode` are automatically set by Dojo. The `domNode` references the top-level node of the widget. So, instead of using `dojo.byId("myId")` you could also use `dijit.byId("myId").domNode`. Other possibilities, using the `dojoAttachPoint` attribute, are described in 9.8.3, “Widget basics” on page 353.

## 9.7.5 Extension points and declarative event connections

You connect events programmatically, for example, with `dojo.connect(widgetDomNode, "onclick", someHandler)`, and also declaratively with Dojo extension points. The two declaration points are `dojo/connect` and `dojo/method`. A usage of the `dojo/connect` is shown in Example 9-49.

*Example 9-49 Declarative event connections with dojo/connect*

---

```
<html>
<head>
<style type="text/css">
    @import "../dijit/themes/tundra/tundra.css";
```



```

</style>
<script type="text/javascript" src="../../dojo/dojo.js"
  djConfig="parseOnLoad: true, isDebug: true"></script>
<script type="text/javascript">
  dojo.require("dijit.form.Button");
</script>
</head>
<body class=tundra>
  <div dojoType="dijit.form.Button" label="Dojo Button">
    <script type="dojo/connect" event="onClick" args="param">
      // param is populated with the event object
      // do something
    </script>
  </div>
</body>
</html>

```

---

Just place the script tag containing `dojo/connect` inside the `div` tag for the widget. The attribute `event` is the event or method to which you want to connect. You can set an `args` attribute if you need access to passed arguments. The value of `args` is a comma-separated list of all arguments. In Example 9-49 on page 340 the value `param` is populated with the event object for the button.

You can use the same syntax with `type=dojo/method`. The main difference between `dojo/connect` and `dojo/method` is that `dojo/method` completely replaces the specified handler, whereas `dojo/connect` only adds another listener to the method passed as the event attribute similar to `dojo.connect`.

## 9.7.6 Instantiating widgets programmatically

Widgets can also be instantiated programmatically. Like classes, you can create a new widget instance with the *new* keyword. Example 9-50 shows how you can instantiate a button widget programmatically.

*Example 9-50 Programmatically instantiating a button widget*

---

```

<html>
<head>
<style type="text/css">
  @import "../../dijit/themes/tundra/tundra.css";
</style>
<script type="text/javascript" src="../../dojo/dojo.js"
  djConfig="parseOnLoad: true, isDebug: true"></script>
<script type="text/javascript">
  dojo.require("dijit.form.Button");

```

```

dojo.addOnLoad(function() {
    // first param is object with attribute map, second the id of the
    // placeholder
    var btn1 = new dijit.form.Button({label: "Dojo Button1"},
        "btn1");
    // second param can also be the placeholder's node
    var btn2 = new dijit.form.Button({label: "Dojo Button2"},
        dojo.byId("btn2"));
    // if you do not have a placeholder, pass a new HTML element and
    // append widget domNode to the body
    var btn3 = new dijit.form.Button({label: "Dojo Button3", id:
        "btn3"}, document.createElement("div"));
    // or do not pass the second param and let Dojo create it
    var btn4 = new dijit.form.Button({label: "Dojo Button4", id:
        "btn4"});
    // append widget's domNode to the body for rendering
    document.body.appendChild(btn3.domNode);
    document.body.appendChild(btn4.domNode);
});
</script>
</head>
<body class=tundra>
    <div id="btn1"></div>
    <div id="btn2"></div>
</body>
</html>

```

---

Check the comments in the example to understand how to instantiate a widget programmatically.

If you want to programmatically instantiate a widget that expects other widgets to be passed, like the `TabContainer`, then some additional effort is needed. Example 9-51 shows how to create the `TabContainer` widget programmatically.

*Example 9-51 Programmatically instantiating a `TabContainer` widget*

---

```

<html>
<head>
<style type="text/css">
    @import "../dijit/themes/tundra/tundra.css";
</style>
<script type="text/javascript" src="../dojo/dojo.js"
    djConfig="parseOnLoad: true, isDebug: true"></script>
<script type="text/javascript">
    dojo.require("dijit.layout.TabContainer");

```

```

dojo.require("dijit.layout.ContentPane");
dojo.addOnLoad(function() {
    var tmp = document.createElement("div");
    document.body.appendChild(tmp);
    var tabCon = new dijit.layout.TabContainer({}, tmp);
    var tab1 = new dijit.layout.ContentPane({title: "Tab1"});
    var tab2 = new dijit.layout.ContentPane({title: "Tab2"});
    tab1.setContent("Some content");
    tab2.setContent("Some content");
    // addChild(child, index), index for specifying order
    tabCon.addChild(tab1); //same as tabCon.addChild(tab1, 0);
    tabCon.addChild(tab2); //same as tabCon.addChild(tab1, 1);
    tabCon.startup();
});
</script>
</head>
<body class=tundra>
</body>
</html>

```

---

All the layout widgets, such as the `StackContainer`, `AccordionContainer`, or `BorderContainer`, can be created in a manner similar to the button widget, shown in Example 9-51 on page 342. Just add the children and then call the `startup` method of the parent to do some initialization. The `startup` method is described in 9.8.3, “Widget basics” on page 353.

### 9.7.7 Custom widget styling

Although the Dojo-provided widgets are visually appealing, their look and feel might not harmonize with other elements on your Web page or you might simply want another theme for the widget. This is not a problem with Dojo.

#### Dijit theme basics

The CSS definitions for all widgets, together with all images that are displayed in widgets, are called a theme. Dojo 1.0 provides the two themes `tundra` and `soria` out-of-the-box. Dojo 1.1 adds a third theme called `nihilo`. Dojo 1.1 also made some changes to the styling for `tundra` and `soria`. To view the new styling run the `dijit/themes/themeTester.html`. Figure 9-17 on page 344 and Figure 9-18 on page 345 show the styling of `tundra` and `soria` for Dojo 1.0 that is included in the Ajax client runtime.

The CSS files are named `tundra.css`, `soria.css`, and `nihilo.css` and are located in the `dijit/themes` folder in their respectively named folders. They all import the

dijit/themes/dijit.css file. The dijit.css file contains common CSS definitions for positioning and sizing. The specific CSS definition file for a specific theme contains the color definitions, setting of background images, and theme-specific positioning and sizing.

There are additional CSS files for bidirectional styling of elements in a right-to-left mode. BiDi styling is discussed in 12.5.11, “Bidirectional (BiDi) layout of elements” on page 594.

## Tundra theme

Figure 9-17 shows the tundra theme.

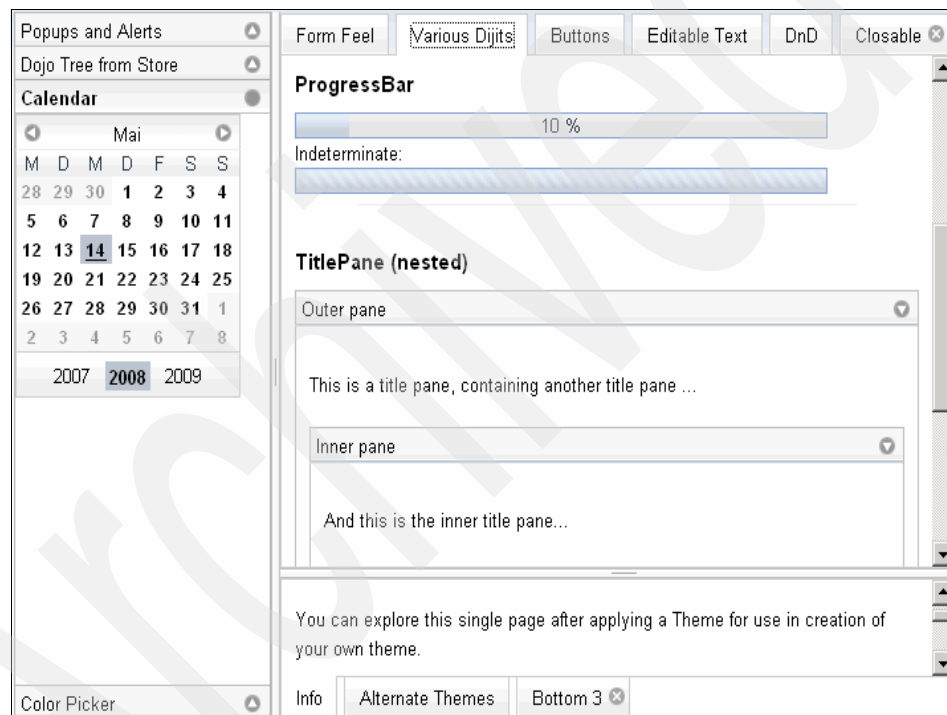


Figure 9-17 Tundra theme in Dojo 1.0

## Soria theme

Figure 9-18 shows the soria theme.

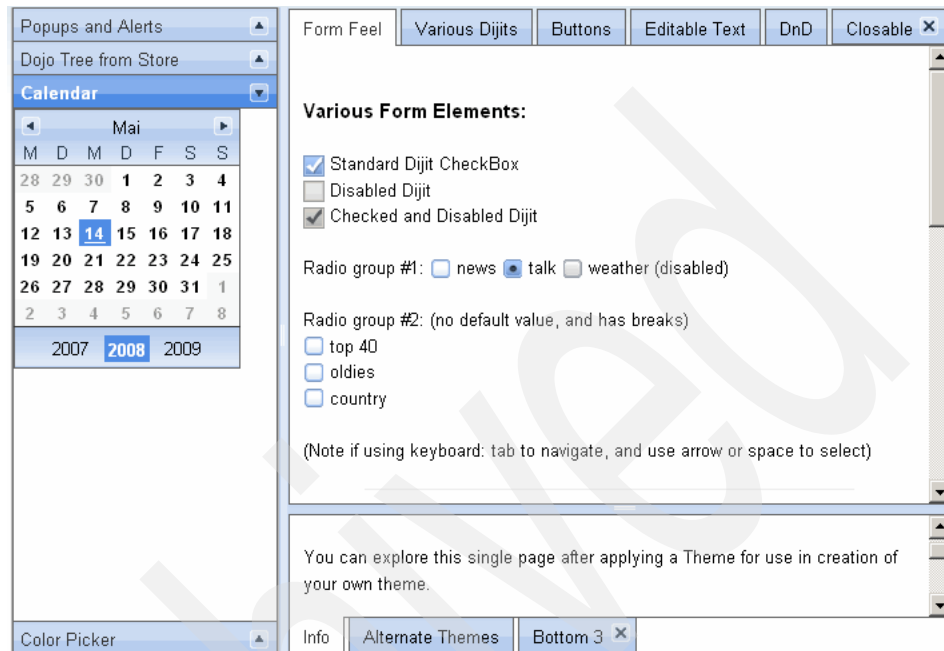


Figure 9-18 Soria theme in Dojo 1.0

The format of the CSS-style definitions for the tundra.css are shown in Example 9-52.

Example 9-52 Format of CSS style definitions in tundra.css

```
.tundra .dijitAlignLeft .dijitTab .dijitClosable .closeImage {
    top: 7px;
    left: 3px;
}

/* SplitContainer */

.tundra .dijitSplitContainerSizerH {
    background:url("images/splitContainerSizerH.png") repeat-y #fff;
    border:0;
    border-left:1px solid #bfbfbf;
```

```
border-right:1px solid #bfbfbf;
width:7px;
}
```

---

The format of the CSS-style definitions for `soria.css` are shown in Example 9-53.

*Example 9-53 Format of CSS style definitions in `soria.css`*

---

```
.soria .dijitVerticalSliderImageHandle {
    border:0px;
    background-position:-400px 0px;
    cursor:pointer;
}

.soria .dijitVerticalSliderBottomBumper {
    border-bottom-width: 1px;
    border-color: #333;
    background: #4f8ce5 url("images/gradientLeftBg.png") repeat-y bottom
left;
}
```

---

To utilize either of these themes you have to manually include it and then set the class attribute of the body tag to either `tundra` or `soria`, respectively, as shown in Example 9-54.

*Example 9-54 Using `tundra.css` style definitions*

---

```
<html>
<head>
<style type="text/css">
    @import "../dijit/themes/tundra/tundra.css";
</style>
<script type="text/javascript" src="../dojo/dojo.js"></script>
</head>
<body class="tundra">
</body>
</html>
```

---

By setting the class name to `tundra` or `soria`, this class name is also inherited by all elements under the body tag and the CSS definitions apply to each element on the Web page.

If you wish to combine both themes, just include both CSS files and then set the class attribute of the elements that require a different theme, as shown in Example 9-55.

*Example 9-55 Using tundra and soria styling together*

---

```
<html>
<head>
<style type="text/css">
    @import "../dijit/themes/tundra/tundra.css";
    @import "../dijit/themes/soria/soria.css";
</style>
<script type="text/javascript" src="../dojo/dojo.js"
    djConfig="parseOnLoad:true"></script>
<script type="text/javascript">
    dojo.require("dijit.form.Button");
</script>
</head>
<body class="tundra">
    <div dojoType="dijit.form.Button" label="Tundra"></div>
    <div dojoType="dijit.form.Button" label="Soria" class=soria></div>
</body>
</html>
```

---

In Example 9-55 the first button has tundra styling. The second button overrides the style and it is rendered with the Soria theme.

## 9.7.8 Overriding Dojo styles

You can write your own theme file and override Dojo's widget styling. However, in many cases you may not want to replace the complete theme but only want to override the styling of specific widgets. In that case, you can either completely override a Dijit style or define your own class. How to override the tundra style is shown in Example 9-56.

*Example 9-56 Overriding Dojo styles*

---

```
<html>
<head>
<style type="text/css">
    @import "../dijit/themes/tundra/tundra.css";
</style>
<style type="text/css">
    .tundra .dijitButtonNode {
        border: 10px ;
    }
</style>
```

```

        background: #FFFF00;
    }
</style>
<script type="text/javascript" src="../../dojo/dojo.js"
    djConfig="parseOnLoad:true"></script>
<script type="text/javascript">
    dojo.require("dijit.form.Button");
</script>
</head>
<body class="tundra">
    <div dojoType="dijit.form.Button" label="Yellow Button"></div>
</body>
</html>

```

---

This example overrides the border and background property for a button widget. All button widgets are now green and have a 10px border.

Note that by defining your own style you are not replacing the complete Tundra styling for the button widget. Only the tundra CSS properties that match with your properties are replaced. Other styles for the ButtonNode class such as padding and vertical-align, defined in the tundra.css file, still apply.

You can also define your own selector `.myClass`, as shown in Example 9-57, and add it to the class of the body.

*Example 9-57 Defining your own CSS selector to override Dojo styles*

---

```

<html>
<head>
<style type="text/css">
    @import "../../dijit/themes/tundra/tundra.css";
</style>
<style type="text/css">
    .myClass .dijitButtonNode {
        border: 10px solid;
        background: #FFFF00;
    }
</style>
<script type="text/javascript" src="../../dojo/dojo.js"
    djConfig="parseOnLoad:true"></script>
<script type="text/javascript">
    dojo.require("dijit.form.Button");
</script>
</head>
<body class="tundra myClass">

```



```
<div dojoType="dijit.form.Button" label="Yellow Button"></div>  
</body>  
</html>
```

---

It does not matter in which order you define the several classes in the class attribute of the body, but the order of loading the CSS definition does matter. If you define your CSS definitions in a separate file, as you would most likely do for a large application, then that file must be loaded after the Dijit theme.

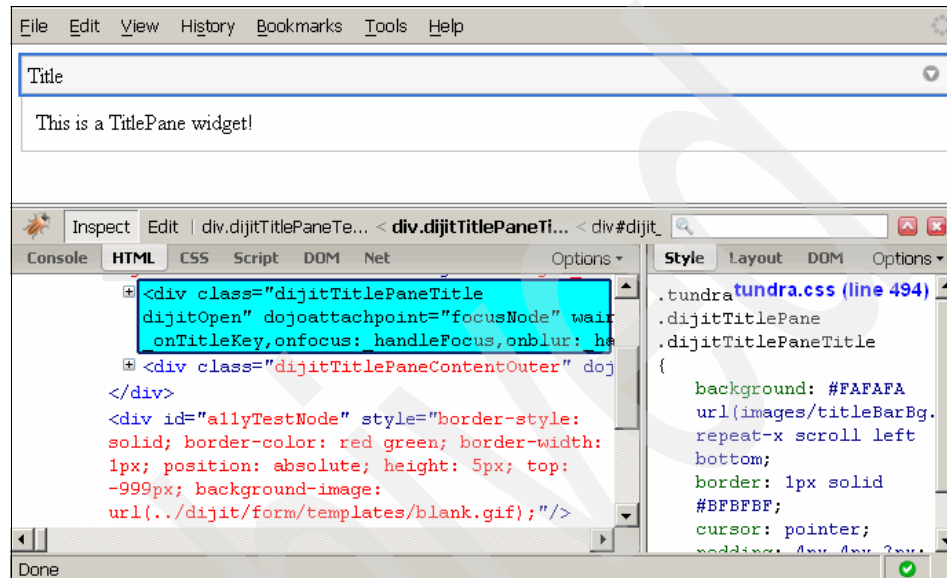
This second approach is more flexible, since you can apply your styles only to specific elements by adding myClass not to the body but to a specific node (as previously shown in Example 9-55 on page 347) for mixing the tundra and soria theme.

### 9.7.9 Finding the correct class to override

Before you can change the style of a widget, you first have to find its respective selectors. You can find the class names by checking the HTML template for the widget and then checking the tundra.css or soria.css files for which properties are set and which ones you need to override. Dojo 1.0 only provides the tundra.css and soria.css. Dojo 1.1 additionally provides a separate CSS file for each widget.

Instead of manually searching for classes in the files, you could inspect the DOM node and all CSS attributes during run time using Firebug to locate the respective classes.

For instance, if you want to change the font size of the title in a TitlePane widget, simply inspect the title bar. Firebug shows the inspected element and you should see the class the you need to override. Figure 9-19 shows the Firebug console. As you see in this example, in order to change the font size of the title bar, you must override the font-size property of the class dijitTitlePaneTitle.



For instance, you might choose a folder structure as shown in Figure 9-20

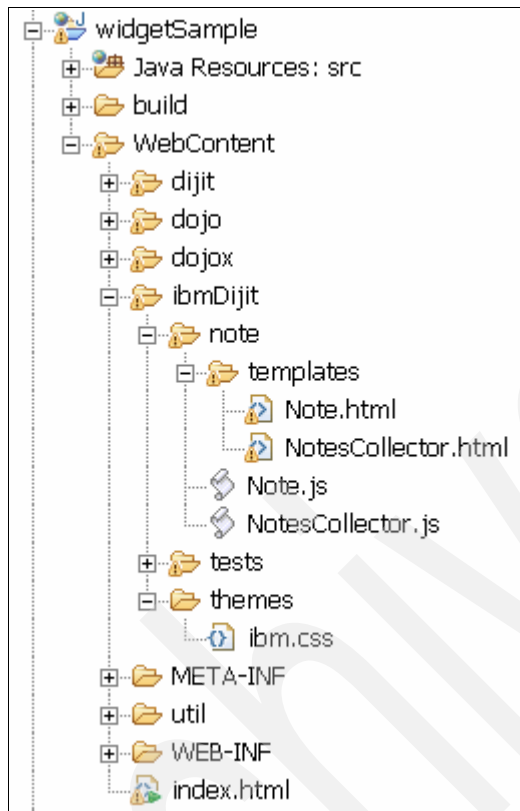


Figure 9-20 Sample widget folder structure

The `ibmDijit` folder structure is similar to the `Dijit` folder. The file `ibm.css` contains all the CSS definitions for the widgets.

Presuming that you choose `.ibm` as a CSS selector for your own styles, an `index.html` file with a basic structure to load your widgets would look as shown in Example 9-58.

Example 9-58 Loading of custom widgets and styles

```
<html>
<head>
<style type="text/css">
    @import "dijit/themes/tundra/tundra.css";
    @import "ibmDijit/themes/ibm.css";
</style>
```

```
<script type="text/javascript" src="dojo/dojo.js"
  djConfig="parseOnLoad:true"></script>
<script type="text/javascript">
  dojo.require("ibmDijit.note.Note");
  dojo.require("ibmDijit.note.NotesCollector");
</script>
</head>
<body class="tundra ibm">
</body>
</html>
```

---

If you do not place your widget folder at the same level as the Dojo directory, then you must set a module path to your directory, as shown in 9.4.2, “Dojo’s packaging system” on page 279.

The folder `ibmDijit/tests` contains unit tests for your widgets. How to write unit tests with Dojo’s Objective Harness (DOH) is described in 9.11, “Unit testing with the Dojo Objective Harness” on page 390.

## 9.8.2 Important functions during the widget life cycle

Before custom widgets are described you should first know which steps are run internally by the Dojo widget framework when you use a widget on your Web page.

Creating a new widget instance either declaratively or programmatically causes Dojo to run specific functions in a specific sequence to initialize internal values before rendering the widget on the Web page.

1. **preamble:** The function `dijit.declare` calls the constructor automatically to control superclass calls. If you need to complete tasks before the superclass constructor is invoked, do it in the preamble. The preamble is called for both the parent and child, but in the reverse order of the constructor, which means first for the child and second for the parent.
2. **constructor:** The constructor is used to add attributes to the widget instance. The attributes are passed as HTML markup (for example, `<div dojoType="dijit.form.Button" label="Hello!">...`) or as a properties object when the widget is created programmatically (for example, `new dijit.form.Button({label:"Hello"})...`). Similar to the sequence used in Java, first the parent constructor is called and then the child constructor.
3. **postMixInProperties:** This is called before nodes of the widget are created and rendered to the page. If your widget template contains placeholders that need to be replaced by dynamically calculated values in the widget JavaScript class, this is the place to do it.

4. `buildRendering`: If the widget is not mixing in `dijit._Templated`, then this function just sets the `domNode` property of the widget. If the widget is used with a template then `buildRendering` is overridden by `dijit._Templated` and the function is responsible for constructing the UI for the widget.
5. `postCreate`: This is called when the DOM node of the widget is created. All placeholders in the widget template are replaced, styling attributes are applied, and the widget is rendered to the Web page. Note that if a widget contains child widgets, the parent's `postCreate` is called before any of the children's `postCreate` functions (that means before any of the child widgets are rendered).
6. `startup`: If the widget contains child widgets, it is called after each child widget is rendered and before any other widgets on the Web page are rendered. It is also called even when the widget does not contain child widgets. For instance, if your page contains an `AccordionContainer` and a button widget is later created using HTML markup, then the calls occur in the following order:
  - a. `postCreate` of `AccordionContainer` (parent)
  - b. `postCreate` of `AccordionPanels` (children)
  - c. `postCreate` of button
  - d. `startup` of `AccordionContainer` (parent)
  - e. `startup` of a button

The `startup` function is called automatically by Dojo when you create the widget declaratively in HTML markup. If you do it programmatically, it is not called and you should call it manually after all children are added to the parent.

Since `preamble`, `constructor`, and `buildRendering` are used for setting up the widget internally and building the main UI for the widget, in most cases you will leave those functions to Dojo and only override `postMixinProperties`, `postCreate`, and `startup` to hook into the life cycle.

Note that compared to the other four functions, `preamble` and `constructor` are called for all classes in the inheritance hierarchy.

### 9.8.3 Widget basics

This section covers the creation of a basic widget to provide an overview of common widget terminology.

The widget that is created is a note widget. It has a title and content, as well as an option to close the note.

## Html template

The HTML template of the note widget is shown in Example 9-59.

*Example 9-59 Note widget HTML template*

---

```
<div class="ibmNote">
  <div class="ibmNoteTitle">
    <div dojoAttachPoint="titleNode" class="ibmNoteTitleNode"></div>
    <div dojoAttachEvent="onclick:_closeNote"
      class="ibmNoteCloseNode" title="{closeTooltip}">X</div>
  </div>
  <div dojoAttachPoint="contentNode" class="ibmNoteContent"></div>
</div>
```

---

The parameters are:

- ▶ **domNode:** The domNode element is the reference to the top-level element in the HTML template. In this case it is the `<div class="ibmNote">` tag. You can access this element with `this.domNode` in the JavaScript part of the widget. The toplevel HTML node is automatically the domNode property. Dojo populates this variable. You can also manually define it in the top-level node.
- ▶ **containerNode:** The attribute containerNode is not used in this example. However, it is described in 9.8.7, “Advanced widgets” on page 366.
- ▶ **dojoAttachPoint:** The dojoAttachPoint is a reference to the element containing that attribute. In this case you can access the element `<div dojoAttachPoint="titleNode" class="ibmNoteTitleNode">` with `this.titleNode`. You can choose the name for the dojoAttachPoint, which must be unique across the widget. This is the main advantage over the use of the id-attribute to access elements with `dojo.byId()`, as the ID must be unique across the entire application.
- ▶ **dojoAttachEvent:** The dojoAttachEvent attribute is used to attach event handlers to elements of the widget. The format is a list of key-value pairs with the event name as the key and the handler function as the value, for example, “onEvent1:\_handler1,onEvent2:\_handler2”. The functions `_handler1`, `_handler2`, and so on, have to be defined in the JavaScript class of the widget. When dojoAttachEvent is not used, you can also attach the handlers manually in the JavaScript class with `dojo.connect()`.
- ▶ **Accessing variables with `{myVar}`:** If you want to dynamically populate the content of an element during startup of the widget, then use the `{myVar}`. The name passed in using `{}` must be defined as a variable in the JavaScript class. An example is the `closeTooltip` variable used in Example 9-59.

## Widget CSS

To make your widget more visually appealing, add class names to the HTML elements and reference them in your CSS file, as shown in Example 9-60.

*Example 9-60 Widget CSS definitions*

---

```
.ibm .ibmNote {
    float: left;
    width: 13em;
    border: 1px solid black;
    padding: 5px;
    font-family: Verdana, Arial, Helvetica, sans-serif;
    background-color: #FFFF00;
    margin: 10px;
}

.ibm .ibmNoteTitle {
    height: 1.3em;
    padding: 5px;
    border-bottom: 1px solid black;
    font-size: 1em;
    font-weight: bold;
}

.ibm .ibmNoteTitleNode {
    float: left;
}

.ibm .ibmNoteCloseNode {
    float: right;
    cursor: pointer;
    border: 1px solid black;
}

.ibm .ibmNoteContent {
    padding: 5px;
    font-size: 0.9em;
}
```

---

## Browser-specific classes

Dojo automatically detects the browser type and version being used and includes the respective classes to the HTML root element of the page. This means that you can easily define CSS selectors that only apply for specific browsers to resolve any browser-dependent styling issues.

The classes that may be set as the value for the class attribute of the HTML root of the page are:

- ▶ `dj_ie`, `dj_ie6`, `dj_ie7`: Internet Explorer. If the browser is determined to be Internet Explorer, `dj_ie` is added as a class. If it is version IE6 then `dj_ie6` is added. If it is version IE7 then `dj_ie7` is added, for example, `<html class="dj_ie dj_ie7">`.
- ▶ `dj_opera`, `dj_opera8`, `dj_opera9`: Opera browser. The same logic applies as for Internet Explorer.
- ▶ `dj_khtml`: KHTML derived browser, for example, Konqueror.
- ▶ `dj_safari`: Safari browser or iPhone.
- ▶ `dj_gecko`: Gecko-based browsers like Firefox or SeaMonkey.

For instance, the style definition `.dj_ie6 .myClass { /* my definition */ }` is only applied for elements with the class `myClass` in Internet Explorer 6.

Dojo 1.1 additionally sets the class `dj_ff2` for Firefox 2.0. For example, if you use Firefox 2.0 then the classes being set are `<html class="dj_gecko dj_ff2">`.

Dojo also adds BiDi-specific and accessibility-specific class names to the body element, which is discussed in 12.5.11, “Bidirectional (BiDi) layout of elements” on page 594, and 12.6.3, “High-contrast mode and turning off images” on page 600.

## Writing the JavaScript code

Now that we have an HTML template we need the widget logic as shown in Example 9-61.

*Example 9-61 Widget JavaScript class*

---

```
dojo.provide("ibmDijit.note.Note");
dojo.require("dijit._Widget");
dojo.require("dijit._Templated");

dojo.declare("ibmDijit.note.Note", [dijit._Widget, dijit._Templated], {
    //define default values for arguments that can be passed when the
    // widget is instantiated
    title: "No title",
    content: "No content",

    // path to the HTML part of the widget, CSS is imported in the main
    // page, e.g. index.jsp
    templatePath: dojo.moduleUrl("ibmDijit.note",
        "templates/Note.html"),
```



```

postMixInProperties: function() {
    // call function from superclass in case other initialization is
    // needed
    this.inherited(arguments);
    // ${closeTooltip} in HTML template is replaced with this value
    this.closeTooltip = "Close";
},

postCreate: function(){
    this.inherited(arguments);
    var a = "";
    //max 10 characters for title
    this.titleNode.innerHTML = this.title.substr(0, 10);
    this.contentNode.innerHTML = this.content;
},

startup: function() {
    this.inherited(arguments);
    // do some other initialization after postCreate
},

// connected with dojoAttachEvent in the HTML template
_closeNote: function() {
    // destroy is inherited from dijit._Widget
    this.destroy();
}
});

```

---

Important attributes that you may need to set or have access to inside your widget class are:

- ▶ **dijit.\_Widget** and **dijit.\_Templated**: The declaration of a widget is similar to a declaration for a normal Dojo JavaScript class. The main difference is that you have to derive your widget class at least from the **dijit.\_Widget** class. This class is responsible for instantiating the widget. If your widget has an HTML template you also have to mixin the **dijit.\_Templated** to instantiate your widget from that template. This class is responsible for building the UI representation out of the template and to populate the **domNode** variable.
- ▶ **templatePath** and **templateString**: The **templatePath** is the path to the HTML template of the widget. You can pass a complete path relative to the Dojo root directory or use **dojo.moduleUrl** if you previously defined a module path. If a simple template is used the template can be passed directly as a string to the

templateString variable, for example, templateString: "<div><div dojoAttachPoint='titleNode'></div></div>".

- ▶ widgetsInTemplate: This attribute is set to true if in addition to the normal HTML elements the HTML template also contains other widgets. The usage is shown in 9.8.7, "Advanced widgets" on page 366.
- ▶ srcNodeRef: This attribute is not used in Example 9-61 on page 356. It is a reference to the HTML element that is used to create an instance of a widget. For instance, if you have <div dojoType="ibmDijit.note.Note"></div>, this.srcNodeRef is a reference to that div element. That element is replaced by the HTML template of the widget.

## Passing arguments to the widget

To make a widget more dynamic you can pass arguments to it as attribute names in the HTML markup or as key-value pairs when instantiating it programmatically.

In order to access the arguments passed to the widget (for example, this.content to access the passed content attribute) you also need to declare them in the widget JavaScript class, as shown in Example 9-61 on page 356.

## Hooking into the widget life cycle

In this example we hooked into the postMixInProperties and postCreate functions. The postMixInProperties is used to populate the variables used in the HTML template, in this case this.closeTooltip, which is referenced by \${closeTooltip} in the HTML template. The postCreate is used to set the title and the content of the Note widget.

If you override one of the life-cycle functions it is a good practice to call the superclass method at the start or end of the function in the event that the superclass needs to perform a required function.

## Usage of example widget

The previously created note widget can be used in the same way as a normal Dojo widget. Example 9-62 shows a declarative usage. The widget can also be used programmatically.

*Example 9-62 Usage of the example note widget*

---

```
<html>
<head>
<style type="text/css">
  @import "ibmDijit/themes/ibm.css";
</style>
<script type="text/javascript" src="dojo/dojo.js"
  djConfig="parseOnLoad:true"></script>
```

```

<script type="text/javascript">
  dojo.require("ibmDijit.note.Note");
</script>
</head>
<body class="ibm">
  <div dojoType="ibmDijit.note.Note" title="My Note"
    content="Do something ... do something ... do something ... do
    something ... do something ... do something ... do something
    ..."></div>
</body>
</html>

```

---

Figure 9-21 shows the output as it is rendered in the browser.

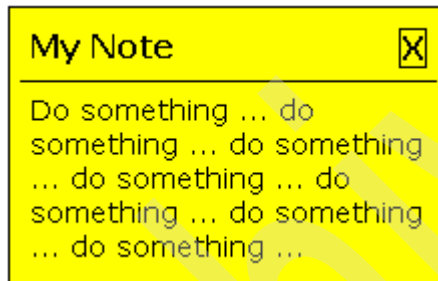


Figure 9-21 Output of the example Note widget

## 9.8.4 Declaring a widget class declaratively

In addition to declaring the widget class programmatically with a separate JavaScript file, you can define a widget declaratively in your HTML with the `dijit.Declaration` class, as shown in Example 9-63.

Example 9-63 Declaration of a widget with `dijit.Declaration`

```

<div dojoType="dijit.Declaration" widgetClass="your.widgetName"
  defaults="{key1:'val1',key2:{key3:'val3'}}"
  mixins="widgetToExtend,mixinClass1,mixinClass2">
  <!-- your HTML template and script tags are added inside -->
</div>

```

---

The parameters are:

- ▶ **widgetClass**: The widget class name that you use to create an instance of this widget.
- ▶ **defaults**: This is a listing of all parameters that can be passed to the widget when creating an instance of it. It is equivalent to the variables that you define in the JavaScript class definition of a widget. You can also pass complex structures, such as an object or an array, as a value in string format.
- ▶ **mixins**: The mixins for the widget. The first string is used as the parent class and the following classes, separated by a comma, are mixed into the widget. The first class must be a subclass of `dijit._Widget` and at least one of the classes should contain a mixin `dijit._Template`.

The template of your widget is defined inside the `dijit.Declaration` tag. Example 9-64 shows the code for a widget that displays a text input box and a button. If you click the button the text in the box is alerted.

*Example 9-64 Usage of the `dijit.Declaration` widget*

---

```
<html>
<script type="text/javascript" src="dojo/dojo.js"
  djConfig="parseOnLoad: true, isDebug: true"></script>
<script type="text/javascript">
  dojo.require("dijit.Declaration");
  dojo.require("dijit.form.Button");
  dojo.require("dojo.parser");
</script>
<style type="text/css">
  @import "dijit/themes/tundra/tundra.css";
</style>
<body class="tundra">
  <div dojoType="dijit.Declaration" widgetClass="ibmDijit.TextAlert"
    defaults="{initVal:'Hello!'}">
    <div>Sample dijit.Declaration</div>
    <input dojoAttachPoint="inputNode" /><div
      dojoType="dijit.form.Button"
      dojoAttachPoint="btnAlert">Alert text!</div>
    <script type="dojo/connect" event="postCreate">
      this.inputNode.value = this.initVal;
      dojo.connect(this.btnAlert, "onClick", this, "alertText");
    </script>
    <script type="dojo/method" event="alertText">
      alert(this.inputNode.value);
    </script>
  </div>
```

```
<div dojoType="ibmDijit.TextAlert" id="yourWidgetId" initVal="Hi
there!"></div>
</body>
</html>
```

---

Imagine the declaration of a widget with `dijit.Declaration` as a block where both the HTML template and JavaScript code are handled in a single block. This means that you have access to the `this`-reference of the widget in a script tag inside the declaration as you would in a separate widget JavaScript file.

The attribute `widgetsInTemplate` is automatically set to `true` when using `dijit.Declaration`.

## 9.8.5 Using the Dojo parser

The Dojo parser was already included in all previous examples. This section describes the different ways in which you can use the parser.

If widgets are included via markup into your HTML page then that markup is converted into the appropriate DOM-representation by the Dojo parser.

### Parsing the entire document

If you want to parse the entire page then set the `parseOnLoad` key in the `djConfig` object to `true`, as shown in Example 9-65. Since the `dojo.parser` module is included in `dijit._Templated`, you do not need to explicitly include it as long as you require at least one widget that includes it. However, you can always include the `dojo.parser` module, since `dojo.require` is intelligent enough to not load a module twice.

*Example 9-65 Automatic parsing of the document by Dojo*

---

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<style type="text/css">
    @import "dijit/themes/tundra/tundra.css";
</style>
<script type="text/javascript">
    djConfig = {parseOnLoad: true};
</script>
<script type="text/javascript" src="dojo/dojo.js"></script>
<script type="text/javascript">
    dojo.require("dojo.parser");
</script>
```

```
</head>
<body class="tundra">
</body>
</html>
```

---

If you use the parser in this way, Dojo will automatically parse the entire page. You do not need to do anything. All the hidden widgets on your Web page will also be parsed.

### **Parsing a specific DOM node**

If you want to have more control of the page parsing in order to implement an individual performance strategy, you can also parse nodes separately.

To parse manually the function `dojo.parser.parse` is used. Instead of setting the `parseOnLoad` value, you can also trigger the parsing of the HTML body manually, as shown in Example 9-66.

*Example 9-66 Manual parsing of the document with the Dojo parser*

---

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<style type="text/css">
    @import "dijit/themes/tundra/tundra.css";
</style>
<script type="text/javascript" src="dojo/dojo.js"
    djConfig="parseOnLoad:true"></script>
<script type="text/javascript">
    dojo.require("dojo.parser");
</script>
<script type="text/javascript">
    dojo.addOnLoad(function() {
        // parses the whole body if no arguments passed
        dojo.parser.parse();
    });
</script>
</head>
<body class="tundra">
</body>
</html>
```

---

Example 9-67 shows how to parse individual HTML nodes.

*Example 9-67 Parsing individual HTML nodes*

---

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<style type="text/css">
    @import "dijit/themes/tundra/tundra.css";
</style>
<script type="text/javascript" src="dojo/dojo.js"
    djConfig="parseOnLoad:false"></script>
<script type="text/javascript">
    dojo.require("dijit.form.Button");
    dojo.require("dojo.parser");
</script>
<script type="text/javascript">
    dojo.addOnLoad(function() {
        var btn1 = dojo.byId("btn1");
        var btn2 = dojo.byId("btn2");

        dojo.parser.parse(btn1);
        //you can also do it with a string that is dynamically created
        btn2.innerHTML = "<div dojoType='dijit.form.Button'
            label='Button2'></div>";
        dojo.parser.parse(btn2);
    });
</script>
</head>
<body class="tundra">
    <div id="btn1">
        <div dojoType="dijit.form.Button" label="Button1"></div>
    </div>
    <div id="btn2"></div>
</body>
</html>
```

---

## 9.8.6 Extending widgets

This section describes techniques to extend a widget with functionally.

## Extending widgets with dojo.extend

Example 9-68 shows how to use the dojo.extend function.

*Example 9-68 Extending a widget with dojo.extend*

---

```
dojo.require("dijit.TitlePane");
dojo.extend(dijit.TitlePane, {
    setTitle: function(/*string*/title) {
        // this overrides the original dijit.TitlePane.setTitle
        // function
    },

    myAddedFunction: function(){
        //added to the widget prototype
    }
});
```

---

The dojo.extend function directly extends the widget's prototype. This means that all instances of dijit.TitlePane that you use on your page use the setTitle function that you defined and have an additional function called myFunction.

## Extending widgets with dojo.mixin

If you do not want to extend the complete widget prototype, use dojo.mixin to extend one instance. You get the same result as for dojo.extend but only for a specific widget instance, as shown in Example 9-69.

*Example 9-69 Extending a widget with dojo.mixin*

---

```
var titlePane = dijit.byId("idTitlePaneInstance");

dojo.mixin(titlePane, {
    setTitle: function(/*string*/title){
        // this overrides the original dijit.TitlePane.setTitle function
        // for this specific instance
    },

    myAddedFunction: function(){
        //added to this specific instance
    }
});
```

---



## Extending widgets with `dojo.declare`

If you do not want to change a widget's prototype, but want all instances of a specific widget to have the extended functionality, then subclass the widget directly.

You can extend a widget in the same way that you did for a normal Dojo class, as shown in 9.4.5, “Object-oriented programming with Dojo” on page 292. Declare your class and derive it from the widget class that you want to extend, as shown in Example 9-70.

*Example 9-70 Extending a widget with `dojo.declare`*

---

```
dojo.provide("ibmDijit.MyTitlePane");
dojo.require("dijit.TitlePane");

dojo.declare("ibmDijit.MyTitlePane", dijit.TitlePane, {
    setTitle: function(title){
        // overrides dijit.TitlePane.setTitle
        // do something
    },

    myAddedFunction: function() {
        // do something
    }
});
```

---

All the rules described in 9.4.5, “Object-oriented programming with Dojo” on page 292, also apply for widget class programming. For example, you can use `this.inherited(arguments)` if you override a method and want to invoke the parent-method, for example, for `postCreate` or `postMixInProperties`.

You can use your extended widget as you would with any widget, as shown in Example 9-71.

*Example 9-71 Usage of widget extensions*

---

```
<html>
<head>
<style type="text/css">
    @import "dijit/themes/tundra/tundra.css";
</style>
<script type="text/javascript" src="dojo/dojo.js"
    djConfig="parseOnLoad:true"></script>
<script type="text/javascript">
    dojo.require("ibmDijit.MyTitlePane");
```

```

dojo.addOnLoad(function() {
    //invoke your added function
    digit.byId("myTitlePane").myAddedFunction();
});
</script>
</head>
<body class="tundra">
    <div dojoType="ibmDijit.MyTitlePane" title="MyTitlePane"
        id="myTitlePane"></div>
</body>
</html>

```

---

The template for your widget subclass is inherited from the template of the widget superclass.

## 9.8.7 Advanced widgets

In this section we develop a more sophisticated widget. The widget is a `NotesCollector`. When instantiated it displays a button. If the button is then clicked a separate layer on top of the current window is faded in. Input fields for a title and the content, along with a button for adding a new note, are displayed. A button for closing the Notes view with a fade effect and returning back to the application are also provided. Figure 9-22 shows how the notes board looks.

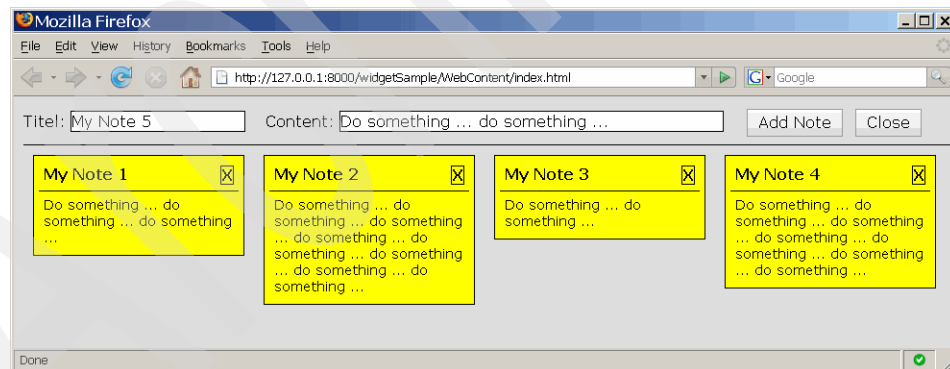


Figure 9-22 *NotesCollector* view

Example 9-72 shows the HTML template for the NotesCollector.

*Example 9-72 HTML template of the NotesCollector widget*

---

```
<div class="ibmNotesCollector">
  <div dojoType="dijit.form.Button" label="${i18n.viewNotes}"
    dojoAttachEvent="onClick:_viewNotes"></div>
  <div dojoAttachPoint="notesControlNode" class="ibmNotesControlNode">
    <div dojoAttachPoint="inputNode" class="ibmNoteInput">
      <label>${i18n.title}</label><input maxlength=10
        class="ibmNotesInputText" dojoAttachPoint="inputTitleNode">
      <label>${i18n.content}</label><input class="ibmNotesInputText
        ibmNotesInputContent" dojoAttachPoint="inputContentNode">
      <div dojoType="dijit.form.Button" label="${i18n.addNote}"
        dojoAttachEvent="onClick:_addNote"></div>
      <div dojoType="dijit.form.Button" label="${i18n.close}"
        dojoAttachPoint="closeBtnNode"></div>
    </div>
    <div dojoAttachPoint="containerNode"></div>
  </div>
</div>
```

---

The elements used are similar to the ones used in the example in 9.7.1, “Widget basics” on page 329. The difference is that this template also contains other widgets and the special Dojo attachpoint `containerNode`. The special property `widgetsInTemplate` that you need to set when your template uses other widgets and the `containerNode` are described further below.

Example 9-73 shows the additional CSS definitions that you need to insert into the `ibm.css` file for styling the NotesCollector widget.

*Example 9-73 CSS definitions for the NotesCollector widget*

---

```
.ibm .ibmNotesCollector {
  font-family: Verdana, Arial, Helvetica, sans-serif;
  font-size: 1em;
}

.ibm .ibmNotesControlNode {
  padding: 10px;
  background-color: #DDDDDD;
  width: 100%;
  height: 100%;
  position: absolute;
  top: 0px;
  left: 0px;
```

```

    }

    .ibm .ibmNoteInput {
        padding-bottom: 5px;
        border-bottom: 1px solid black;
    }

    .ibm .ibmNotesInputText {
        margin-left: 5px;
        margin-right: 15px;
        border: 1px solid black;
        font-family: Verdana, Arial, Helvetica, sans-serif;
        font-size: 1em;
    }

    .ibm .ibmNotesInputContent {
        width: 25em;
    }

```

---

Example 9-74 shows the JavaScript class for the NotesCollector widget.

*Example 9-74 JavaScript class of the NotesCollector widget*

---

```

dojo.provide("ibmDijit.note.NotesCollector");
dojo.require("dijit._Widget");
dojo.require("dijit._Templated");
dojo.require("dijit.form.Button");
dojo.require("ibmDijit.note.Note");

dojo.declare(
    "ibmDijit.note.NotesCollector", [dijit._Widget, dijit._Templated], {
        templatePath: dojo.moduleUrl("ibmDijit.note", "
            templates/NotesCollector.html"),

        // parse the template to find widgets declared in markup inside it
        widgetsInTemplate: true,

        postMixInProperties: function() {
            // you can obtain an internationalized string at this point
            this.i18n = this._getI18NStrings();
            // call function from superclass in case other initialization is
            // needed
            this.inherited(arguments);
        },
    },

```

```

postCreate: function(){
    // instead of using dojoAttachEvent you can also connect the
    // button pogramatically
    dojo.connect(this.closeBtnNode, "onClick", this, this._close);
    this._fade(false, 0);
    this.inherited(arguments);
},

_getI18NStrings: function() {
    return {
        "addNote": "Add Note",
        "title": "Titel",
        "content": "Content",
        "viewNotes": "View Notes",
        "close": "Close"
    }
},

_viewNotes: function() {
    this.notesControlNode.style.display = "";
    this._fade(true, 1000);
},

_addNote: function() {
    var newNote = new ibmDijit.note.Note({"title":
        this.inputTitleNode.value,
        "content": this.inputContentNode.value});
    this.containerNode.appendChild(newNote.domNode);
},

_close: function() {
    // this.notesControlNode.style.display = "none";
    this._fade(false, 1000);
},

_fade: function(isIn, duration) {
    dojo[isIn ? "fadeIn" : "fadeOut"]({
        node: this.notesControlNode,
        duration: duration,
        onEnd: dojo.hitch(this, function() {
            this.notesControlNode.style.display = isIn ? "" : "none";
        })
    });
}

```

```

        })
    }).play();
}
});

```

---

If you use widgets inside your HTML template, you additionally have to set the Dojo variable `widgetsInTemplate` to the value `true` to allow Dojo to also parse those widgets. Parsing widgets takes more time than parsing normal HTML elements. Thus, for performance reasons, you should only set `widgetsInTemplate` to `true` if the parsing is really needed. The default value is `false`.

The class also shows a technique of how to internationalize your strings in the widget. Define the strings used in the template with the `${varName}` construct and populate those variables in the `postMixInProperties` function. This example populates the `i18n` variable with static strings. See 12.5.1, “Dojo i18n” on page 580, for how to read translated strings from resource files.

Like the attachpoint `domNode` of a widget, the attachpoint `containerNode` is automatically populated by Dojo as well. It contains all elements that are passed to the widget when it is instantiated. Dojo automatically adds the elements that are passed inside the `containerNode` element.

Example 9-75 shows how you can instantiate a `NotesCollector` widget that already contains three notes when the `NotesCollector` view is opened.

*Example 9-75 Passing element to the containerNode attachpoint*

---

```

<html>
<head>
<style type="text/css">
    @import "dijit/themes/tundra/tundra.css";
    @import "ibmDijit/themes/ibm.css";
</style>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="parseOnLoad:true"></script>
<script type="text/javascript">
    dojo.require("ibmDijit.note.Note");
    dojo.require("ibmDijit.note.NotesCollector");
</script>
</head>
<body class="tundra ibm">
    <div dojoType="ibmDijit.note.NotesCollector">
        <div dojoType="ibmDijit.note.Note" title="My Note 1" content="Do
            something ... "></div>

```

```
<div dojoType="ibmDijit.note.Note" title="My Note 2" content="Do
  something ... "></div>
<div dojoType="ibmDijit.note.Note" title="My Note 3" content="Do
  something ... "></div>
</div>
</body>
</html>
```

---

## 9.9 Data API

The `dojo.data` module provides Dojo applications with a standardized mechanism for accessing data resources. It is an abstraction layer for uniform access to such resources, called data sources, from a Dojo application.

Data sources refer to the source of the raw data. This could be a file (that is, `.csv`, `.txt`, `.json`), a database server, a Web service, or a variety of other types. Since the `dojo.data` module accesses all data sources uniformly, the data source can be as simple or as complex as needed, without affecting how it is used.

Datastores are the essence of the `dojo.data` module. A datastore is a JavaScript object that acts as a layer between data being read from a data source and then makes that data available through a standard set of APIs. The data is represented as an item or as an attribute of an item. `Dojo.data` is flexible in that it allows APIs to be created for custom data formats or limited functionalities (that is, read-only custom data-type datastores). Datastore-aware widgets can then access the custom data through the standard APIs without having to be knowledgeable about the specific API format needed to access the custom data.

What this means is that the raw data structure (be it JSON, XML, CSV, or anything else) is hidden, therefore allowing the data to be accessed in a consistent manner, regardless of whether the data access is handled on the client or server side. The responsibility for interpreting and presenting the data in a common access format is transferred from the business-logic code to the store.

### 9.9.1 Core design concepts

There are a number of fundamental design concepts behind the `dojo.data` module that must be understood to utilize it fully.

## 9.9.2 Datastores and dojo.data APIs

Dojo.data is not a single API. It has been separated into distinct APIs, as shown in Table 9-3.

Table 9-3 The basic dojo.data APIs

Name	Description
dojo.data.api.Read	Reading data items and item attributes. The Read API also allows data items to be filtered, searched, and sorted.
dojo.data.api.Write	The Write API deals with the creation, deletion, and modification of items and their attributes.
dojo.data.api.Identity	Data items may have an identifier that acts as a unique index. The Identity API allows items to be located and fetched using their identity value.
dojo.data.api.Notification	The Notification API notifies listeners of a datastore about change events on the store's data items (for example, when items are created, deleted, or modified).
dojo.data.api.Request	The Request API outlines request objects returned from fetch() methods. The objects contain the query parameters as well as an abort() function.

Not all data sources can provide all the features provided by the above four APIs combined, hence the need for their separation. Therefore, a store can choose to implement all or a subset of the functionalities listed above. To provide detailed information about what functions a datastore implementation provides, each store must provide a `getFeatures()` function, which reports which of the above APIs the store implements.

A datastore's `getFeatures()` function should return a map, the keys of which represent the APIs that the store implements. The value for these keys should be something sensible like `true` or, if appropriate, a logical value (that is, the name of the identity attribute if the store implements the Identity API).



The Dojo Ajax client runtime that ships with IBM WebSphere Application Server Feature Pack for Web 2.0 contains the datastore implementations given in Table 9-4.

*Table 9-4 The basic dojo.data stores*

Name	Description
dojo.data.ItemFileReadStore	Store for JSON data; read-only
dojo.data.ItemFileWriteStore	Store for JSON data; read/write

These stores accept as input a JavaScript object that has the structure given in Table 9-5.

*Table 9-5 Wrapper object structure for ItemFile\*Stores*

Name	Description
identifier	Optional; meta data that specifies which attribute of each item in the store will act as the item's unique identifier.
label	Optional; meta data that specifies which attribute of each item in the store will act as the item's readable descriptor.
items	Required; JavaScript object array that will act as the store's items. Can use <code>_reference</code> to point to other item objects (see "References" on page 385).

The stores' constructors takes parameters as keyword arguments, as shown in Table 9-6.

*Table 9-6 Constructor parameters for ItemFile\*Stores*

Name	Description
url	Optional*; the JSON data URL; if not used, use data.
data	Optional*; the JavaScript object described above. If not used, use url.
typeMap	Optional; a JavaScript map of data types used for deserialization (see "Custom data types" on page 386).

An example of a client-side JSON data-based store defined programmatically is shown in Example 9-76.

*Example 9-76 Example data store: programmatically*

---

```
var defaultData = {
  "identifier": "id",
  "label": "name",
  "items": [
    { "id": 1, "name": "Sarah P.", "location": "NYC" },
    { "id": 2, "name": "Amy H.", "location": "London" },
    { "id": 3, "name": "Gwen H.", "location": "London" },
    { "id": 4, "name": "Hugh O'B.", "location": "Dublin" },
    { "id": 5, "name": "Brian B.", "location": "Dublin" }
  ]
};

var myStore = new dojo.data.ItemFileWriteStore({data:defaultData});
```

---

An example of a client-side JSON data-based store defined declaratively is shown in Example 9-77.

*Example 9-77 Example data store: declaratively*

---

```
// using the same default data as above
<div id="myStore" dojoType="dojo.data.ItemFileWriteStore"
    jsId="myStore" data="defaultData">
</div>
```

---

**Note:** The previous technique of using a local data object is both a useful and a flexible method for developing the client-side while the server-side is also being developed. The *pluggable* nature of datastores means that for development, a local store could be used for production, and later a reference to a custom datastore replaces the local data:

```
var myStore = new dojo.data.ItemFileWriteStore({url:"people.json"});
```

Also, in the previous example, when programmatically creating a store using the data argument, the defaultData object is modified and directly used by the datastore. That is because the store keeps a representation of the data in its own format for efficiency. This means that if you want to re-use defaultData, you may find that it has been modified and no longer is in the format that you expected. To circumvent this behavior, you could clone the object and pass the clone instead, or alternatively switch to a URL pointing to a local file containing the same data.

The Feature Pack for Web 2.0 also comes prepackaged with stores for more complex formats. The stores shown in Table 9-7 are part of the DojoX data project.

Table 9-7 The DojoX datastores

Name	Description
dojox.data.CsvStore	Store for comma-separated variable (CSV) formatted data (for example, spreadsheet data) (read-only).
dojox.data.FlickrStore	Store for queries on the online image Web service flickr.com (read-only).
dojox.data.FlickrRestStore	Store for queries on the online image Web service flickr.com using a REST-based approach (read-only).
dojox.data.HtmlTableStore	Store for HTML table formatted data.
dojox.data.OpmlStore	Store for Outline Processor Markup Language (OPML) (read-only).
dojox.data.PicasaStore	Store for queries on Google's online image Web service Picasa (read-only).
dojox.data.QueryReadStore	Store for JSON data (read-only). Queries server on each request.
dojox.data.XmlStore	Store for basic XML data (read/write).

## Accessing data item attribute values

To access data item attribute values in a datastore, `dojo.data` does not use the traditional JavaScript method for working directly with the data item. For instance, the following are incorrect:

```
// traditional method
value = item.attribute;

// alternative method
value = item["attribute"];

// using getValue()
value = item.getValue("attribute");
```

All items and their attributes should be accessed through their containing store. It is the store, not the item, that controls and manages the interactions with item values.

```
// correct method
value = myStore.getValue(item, "attribute");
```

The benefits of accessing data this way are:

- ▶ Having a standardized method for data value access, such that the internal structure of the data is abstract from the application using it.

This means that the data can remain in its most efficient format, that no resources are needed to convert that data to a particular defined format, and that the structure may even be compacted internally without affecting how that data is used in a Dojo application.

It also allows for a consistent and pleasing mechanism for interacting with a myriad of data types.

- ▶ As the store is the manager of data access, it may also control how that data is delivered to the application, including using the technique of lazy-loading (useful for large datasets; see “Lazy loading” on page 387).

The above points demonstrate that using the store as the accessor provides much greater options for flexibility and performance for the user.

### **Asynchronously fetching items**

As will be described in later sections, the `fetch()` function (from the `dojo.data.util.simpleFetch` mixin) and its related `fetchItemByIdentity()` (should the store use the Identity API) function work asynchronously.

The reason for this is that not all stores work synchronously by default and not all stores will have their data organized in an ordered, sequential format. By being asynchronous, we allow enhanced flexibility and callbacks like `onItem` (the callback function is called for each item retrieved) and `onComplete` (when the fetch has completed) to be used to provide a solution adaptable to a variety of needs.

### 9.9.3 Read API

The Read API is the interface at the heart of `dojo.data`. All datastores must implement this API since every store needs the core item retrieval and interaction functionality. Following are descriptions of these functions:

- The datastore can detail which Data APIs it implements through `getFeatures()`, as shown in Example 9-78.

*Example 9-78 getFeatures()*

---

```
var myStore = new com.ibm.itso.customDatastore();
var storeFeatures = myStore.getFeatures();
dojo.forEach(storeFeatures, function(feature, index){
    console.log("Store implements feature: " + feature);
});
```

---

- Detail the attributes of each item in a datastore regardless of the underlying data format, as shown in Example 9-79.

*Example 9-79 Output an item's attributes*

---

```
var myStore = new com.ibm.itso.customDatastore();
// myItem has been fetched
var itemAttributes = myStore.getAttributes(myItem);
dojo.forEach(itemAttributes, function(attr, index){
    console.log("Item Attribute: " + attr);
});
```

---

- Get the value of an attribute regardless of the underlying data format, as shown in Example 9-80.

*Example 9-80 Get value of item attribute*

---

```
var value = myStore.getValue(myItem, "myAttribute");
```

---

- Check whether a JavaScript object is an item in the datastore, as shown in Example 9-81.

*Example 9-81 Checking whether an item is in the store*

---

```
if(myStore.isItem(myJSObject) {
    console.log("Object is an item in the store.");
} else {
    console.log("Object is NOT an item in the store.");
}
```

---

- Fetch all the items from a store, as shown in Example 9-82.

*Example 9-82 Fetching all store items*

---

```
// onComplete callback function
var myStoreItemsFetched = function(items,request) {
    console.log("Found " + items.length + " items.");
};

myStore.fetch({onComplete: myStoreItemsFetched});
```

---

The Read API file is in the Ajax client runtime supplied with the feature pack:

`dojo.data.api.Read.js`

## 9.9.4 Write API

Datastores that implement the Write API provide the ability to save new items to the datastore, modify existing items, and remove items from the store.

Stores that implement the Write API will use a two-phase approach for interaction with the eventual service that manages the persistence. Two phases permits techniques like operation batching so that a group of items may be persisted in one function call.

Descriptions of these functions follow:

- Adding items is shown in Example 9-83.

*Example 9-83 Adding an item to a datastore*

---

```
myStore.newItem({
    "id": someData.index,
    "name": "Jane H.",
    "location": "Limerick"
});
```

---

The `newItem` function takes a `keywordArgs` parameter. The recommended usage is that the argument should be a JavaScript object that matches the structure of the item's attribute list. The store implementation is responsible for handling any needed conversion to a specific data format (that is, XML).

- Modifying items is shown in Example 9-84.

*Example 9-84 Modifying an item's attributes*

---

```
// myItem has been fetched previously
myStore.setValue(myItem, "myAttribute", "new attribute value");
```

---

- Saving items is shown in Example 9-85.

*Example 9-85 Saving a modified item*

---

```
// the items that are dirty
dojo.forEach(items, function(item, index) {
    if(myStore.isDirty(item)) {
        console.log("Item at index " + index + " is dirty.");
    }
});

// a store is dirty if the in-memory representation of the store is
// inconsistent with the actual store data i.e. an existing item
// has been modified or a new item has been added to the store
if (myStore.isDirty()) {
    myStore.save();
}
```

---

Saving items is asynchronous.

- Deleting items is shown in Example 9-86.

*Example 9-86 Deleting an item from the datastore*

---

```
myStore.deleteItem(itemToBeDeleted);
```

---

- Reverting to a previous state is shown in Example 9-87.

*Example 9-87 Reverting a store to the last save point*

---

```
if (myStore.isDirty()) {
    myStore.revert();
}
```

---

Reverting to a previous state applies to store items on the client side.

The datastore tracks all `newItem`, `deleteItem` and `setValue` calls on items for batch saving and for reverting.

The Write API file is in the Ajax client runtime supplied with the feature pack:

`dojo.data.api.Write.js`

## 9.9.5 Identity API

Datastores can implement the Identity API if a data source provides a method for data items to be uniquely identified. When an item has a unique identity, a search using a specific attribute value is not required to find the item.

Identifiers should be unique and apply to only one item in the datastore. Identifiers may also be compound keys, as long they can be converted to a string using the `toString()` JavaScript API.

Descriptions of these functions follow:

- Finding items by their identifier is shown in Example 9-88.

---

*Example 9-88 `fetchItemByIdentity()`*

---

```
var myStore = new dojo.data.ItemFileWriteStore({data:defaultData});

function myOnItem(item) {
    if (myStore.isItem(item)) {
        console.log("Name: " + myStore.getValue(item, "name"));
    }
}

// asynchronously find the item with id of 3
myStore.fetchItemByIdentity({identity: 3, onItem: myOnItem});
```

---

- Getting an item's identity is shown in Example 9-89.

---

*Example 9-89 `Displaying all the items with their identifier value`*

---

```
var myStore = new dojo.data.ItemFileWriteStore({data:defaultData});

function myOnComplete(items,request) {
    for (var i = 0; i < items.length; i++) {
        console.log("Item " + i + " identity value: " +
myStore.getIdentity(items[i]));
    }
}

// find all items
myStore.fetch(onComplete: myOnComplete);
```

---

- Displaying which attributes comprise the identifier.

Datastores that implement the Identity API may expose the identity either as a public or a private attribute of the item. If it is public, it must return the attributes identifier from the `getIdentityAttributes()` method. Otherwise, this



method can return null. Displaying an item's actual identifier attributes is shown in Example 9-90.

*Example 9-90 Displaying an item's actual identifier attributes*

---

```
var myStore = new dojo.data.ItemFileWriteStore({data:defaultData});

function myOnError(error) {
    console.log("There was an error: " + error);
}

function myOnItem(item) {
    if (myStore.isItem(item)) {
        var idAttrs = myStore.getIdentityAttributes(item);
        for (var i = 0; i < idAttrs.length; i++) {
            console.log("Identity Attribute " + i + ": " + idAttrs[i]);
        }
    }
}

// fetch an item
myStore.fetchItemByIdentity({identity: 4, onItem: myOnItem, onError:
myOnError});
```

---

The Identity API file is in the Ajax client runtime supplied with the feature pack:  
dojo.data.api.Identity.js

## 9.9.6 Notification API

The Notification API provides monitoring of the basic events (creation, deletion, and modification) affecting items in a datastore.

### Comparison with dojo.connect()

As described in 9.4.8, “Dojo's event system” on page 302, Dojo has a pattern for listening for event changes on an object. The `dojo.connect()` event model can be bound to the `onNew`, `onDelete`, and `onSet` functions on the datastore and will invoke your function whenever the store calls any of those methods. You can choose to replace the actual notification functions themselves with your own custom implementation. When the store then calls the notification functions, your logic will then be invoked.

When a store implements the Identity API, *all* change events on items result in a call to the respective notification function. Although the notifications occur at the

level of the datastore, you can add granularity by implementing your own event handling (that is, only tracking certain event types or particular items).

The Notification API is shown in Example 9-91.

*Example 9-91 The Notification API*

---

```
var myStore = com.ibm.itso.CustomNotificationStore({data:defaultData});

myStore.onNew = function(item) {
    alert("Created item: " + this.getLabel(item));
}

myStore.onSet = function(item, attribute, oldValue, newValue) {
    alert("Item " + this.getLabel(item) + " modified; attribute changed
was " + attribute: old = " + oldValue + ", new = " + newValue);
}

myStore.onDelete = function(item) {
    alert("Deleted item: " + this.getLabel(item));
}

// trigger onNew alert
myStore.newItem({
    "id": 6,
    "name": "Randal H.",
    "location": "Limerick"
});

// trigger onSet alert
myStore.fetchItemByIdentity({identity: 6,
    onItem: function(item) {
        myStore.setValue(item, "location", "Newcastle West");
    }
});

// trigger onDelete alert
myStore.fetchItemByIdentity({identity: 6,
    onItem: function(item) {
        myStore.deleteItem(item);
    }
});
```

---

A recommended performance pattern for stores implementing the Notification API is to implement these functions as simple no-op bind points. Should you wish to add your own custom notification implementation, you can easily do so.

The Notification API file is in the Ajax client runtime supplied with the feature pack:

```
dojo.data.api.Notification.js
```

## 9.9.7 Query

So far we have used `fetch()` to return a list of items in the datastore and worked with the individual items by looping through them. However, Dojo's Data APIs do provide functionality for directly selecting a subset of items.

A query is a JavaScript object and its attributes will mimic those of the items in the datastore. The query object is passed to the `fetch` with a name-value pair consisting of an attribute and value such that the query object's attributes and values act as a type of regular expression for the store's items' attributes and their values to query against. For instance, if we were to query for the location Dublin, our query object would be:

```
{ "location": "Dublin" }
```

Since each type of datastore may have its own query syntax, we recommend using, like the basic `ItemFile*Store` stores, the traditional query syntax that is present in Java, UNIX shell scripts, and so on. That is, to use the wildcards (\*) (representing any characters) and question marks (?) (representing a single character). For example:

```
{ "location": "Dub*" } // will find Dublin
```

and

```
{ "location": "Lond?n" } // will find London
```

Multiple attributes imply an AND function between the terms:

```
// will match a person, for example, with the name "Melanie S."  
// AND location of "Munich"  
{ "name": "*Melanie*", "location": "Mun*" }
```

You can now pass the query along with the `fetch()` (Example 9-92).

*Example 9-92 fetch() with a query*

---

```
myStore.fetch({  
  query: { location: "Dubl*" },  
  onComplete: function(items, request) {
```

```
        for (var i = 0; i < items.length; i++) {  
            if (myStore.getValue(item[i], "location") === "Dublin") {  
                console.log("Cool, the query worked!");  
            }  
        }  
    }  
});
```

---

Queries also have options that affect the behavior of the query as opposed to the actual search terms. As queries are by default case-sensitive and search only the single horizontal level of a hierarchy, they can be altered to ignore case and perform deep searches by using the `queryOptions` argument.

*Example 9-93 fetch() with a query and with queryOptions*

---

```
myStore.fetch({  
    queryOptions: { ignoreCase: true, deep: true },  
    query: { location: "LIMeriCK" },  
    onComplete: ...  
});
```

---

## 9.9.8 Request API

Request objects are returned from the `fetch()` function. The object will contain the query parameters supplied to the `fetch()` function along with an additional `abort()` function. Aborts are hook points for stopping long-running fetches or to halt streaming callbacks from a fetch.

Stores can also use the request object to place information about the request in process (that is, how to use the store's internal cache). A good example of re-using a request object is for pagination (see "Pagination" on page 387).

## 9.9.9 Data formats

The sample data used so far has been a simple flat structure. If we want to use a more complex structure we can use hierarchies of nested items or use the references for the child nodes to act as pointers to other data objects. For example, the basic datastores that ship with the Ajax client runtime, `dojo.data.ItemFileReadStore` and `dojo.data.ItemFileWriteStore`, can use the following data formats discussed in this section.

## Nested items

JSON data with a hierarchy of nested items is shown in Example 9-94.

*Example 9-94 JSON data with hierarchy of nested items*

---

```
{
  "identifier":"name",
  "label":"name",
  "items": [
    { "name":"TCD", "type":"University", "children": [
      { "name":"Science and Engineering", "type":"Faculty",
"children": [
        { "name":"Chemistry", "type":"School" },
        { "name":"Engineering", "type":"School", "children": [
          { "name":"Computer Science", "type":"Dept" } ]
        } ]
      }
    ],
    { "name":"MIT", "type":"University", "children": [
      { "name":"Arts", "type":"School", "children": [
        { "name":"English", "type":"Dept" } ]
      },
      { "name":"Business", "type":"School" } ]
    }
  ]
}
```

---

## References

JSON data with references is shown in Example 9-95.

*Example 9-95 JSON data with references*

---

```
{
  "identifier":"name",
  "label":"name",
  "items": [
    { "name":"TCD", "type":"University",
      "children": [
        { "_reference":"Science And Engineering" } ]
    },
    { "name":"MIT", "type":"University",
      "children": [
        { "_reference":"Arts" },
        { "_reference":"Business" } ]
    },
    { "name":"Science And Engineering", "type":"Faculty",
```

```

    "children": [
      { "_reference": "Chemistry" },
      { "_reference": "Engineering" } ]
  },
  { "name": "Chemistry", "type": "School" },
  { "name": "Engineering", "type": "School",
    "children": [
      { "_reference": "Computer Science" } ]
  },
  { "name": "Arts", "type": "School",
    "children": [
      { "_reference": "English" } ]
  },
  { "name": "Business", "type": "School" },
  { "name": "Computer Science", "type": "Dept" },
  { "name": "English", "type": "Dept" } ]
}

```

## Custom data types

Custom data types can be used to instruct the store as to how it should handle certain types of attribute values. For ItemFile\*Store, the format required is shown in Table 9-8.

Table 9-8 Custom data types

Name	Description
_type	Looks up the typeMap for which constructor to use based on this attribute value
_value	Parameter to be passed to the constructor that was looked up for this type

ItemFile\*Store already has a custom data type of the date object. If the attribute value supplied is { \_type: "Date", \_value: "an ISO String" } then it is treated as a new date object and the ISO string is passed to the date's deserialize function, rather than it being treated as a new JavaScript object with two attributes. This is accomplished by general-purpose mapping, as can be seen in dojo.data.ItemFileReadStore.js:

```

// If no default mapping for dates, then set this as default.
// We use the dojo.date.stamp here because the ISO format is the
// 'dojo way' of generically representing dates.
this._datatypeMap['Date'] = {
  type: Date,
  deserialize: function(value) {

```

```

        return dojo.date.stamp.fromISOString(value);
    }
};

```

A simpler implementation is to map directly to the constructor by using the `_value` attribute to supply any constructor parameters needed:

```

var myStore = new dojo.data.ItemFileWriteStore({
    data:defaultData,
    typeMap: {
        "custom":some.Custom,
        ...
    }
});

```

## Lazy loading

Items can be loaded in a lazy manner by use of the store's `isItemLoaded()` and `loadItem()` methods. For instance, using the data in Example 9-95 on page 385, you could fetch all the items that had a type of university and then loop through its attributes, first testing whether there are any items and, if so, testing whether the item is loaded and, if not, loading it.

## Pagination

For large data sets it may be too much work for the client-side to handle all the returned results of a `fetch()`. For this situation, Dojo.data provides a way for a store to load just a subset of the available data and then using the Request API (9.9.8, “Request API” on page 384) re-use the returned request object to fetch the next batch of results:

```

request = myStore.fetch({onComplete: onMyComplete, start: 0, count:
numItemsToFetch});

```

The `fetch()` can take a `start` parameter (if omitted we assume it to be 0), which informs the store of which index in the collection of items to start returning items, and a `count` parameter (if committed it returns all items), which indicates how many items to return. The request object returned from the fetch can then be reused by giving new values for the `start` and `count` parameters:

```

// go forward by numItemsToFetch amount
request.start += numItemsToFetch;
myStore.fetch(request);

// go back by 50 items;
request.start -= 50;
myStore.fetch(request);

```

If you expect to be working with large data sets, we recommend that you implement your own store and have the server-side transfer large batches of data to the client application and then handle the pagination yourself. Stores, like `ItemFileReadStore`, attempt to load all the data available from the server at once and only use pagination on the client-side. Therefore, you should first know how your application will work with data sets and then tailor your datastore implementation to your needs.

## Sorting

Depending on the store implementation being used, the order returned from `fetch` cannot be assured. The API does, however, provide a mechanism for sorting the items returned by the `fetch` in a datastore. You pass an array of the item attributes to sort as well as the direction of the sort to the `fetch` method as a parameter:

```
var sortArray = [ {attribute:"type", descending:false} ];
```

```
myStore.fetch({
```

The array order dictates sort priority:

```
var sortArray = [  
    { attribute:"name", descending:"true" },  
    { attribute:"type", descending:"false" } ];
```

## 9.10 Dojo extensions in dojoX

DojoX is a set of subprojects that extend the Dojo toolkit. The subprojects can be independent of or dependent on other Dojo and Dijit packages or DojoX projects itself. When a subproject becomes stable it is then a candidate for being moved into Dijit or Dojo core packages based on the needs of the toolkit. Unsuccessful subprojects are moved out of the DojoX package. DojoX subprojects are not guaranteed to be internationalized or fully accessible. For more information, check the `README.txt` file in each of the DojoX projects in the `dojoX` folder.

### List of DojoX projects

The following list gives a brief description of some of the current DojoX projects:

- ▶ **Comet:** A client implementation of the Bayeux protocol to connect to comet-enabled servers. A server-side implementation is provided by the WebSphere Feature Pack for Web 2.0 and is described in 10.5, “Cometd and the Dojo Toolkit” on page 465.
- ▶ **Charting:** A simple charting library based on the Fx and Gfx projects in DojoX



- ▶ Collections: Provides functionality to handle complex collection types like ArrayLists or dictionaries.
- ▶ Color: Extended functionality for color processing. Dependent on the Dojo color classes.
- ▶ Data: Extensions and additional data stores implementing the dojo.data API. These include a data store interface to the public photo feed services of Flickr and Picasa, a data store for XML-based services, along with other useful data stores.
- ▶ Date: Provides extended functionality for date formatting functions common to other languages like PHP or those derived from POSIX.
- ▶ Django template language (Dtl): An implementation of the Django template language to easily compile a template once and then render it indefinitely.
- ▶ Encoding: Provides functionality for common encoding algorithms.
- ▶ Flash: Provides functionality for high-performance JavaScript-Flash communication.
- ▶ Fx: Additional animation affects extending the base effects of Dojo core.
- ▶ Gfx and Gfx3d: A powerful 2D graphics API and an implementation of a portable 3D graphics library.
- ▶ Grid: In HTML terms the grid widget is a super-table. It has the following main features:
  - Lazy loading of rows (100,000+ rows)
  - Rich in-place editing of cells
  - Addition or deletion of rows
  - Loading data from data stores with the dojo.data API
  - Resizing of cells
- ▶ Image: Widget collection for presenting pictures on the window.
- ▶ Layout: Extensions for Dijit's layout widgets.
- ▶ Math: Provides advanced math functions, abstract curve definitions, and some point calculations.
- ▶ Presentation: A widget for various display-oriented tasks. Enables presentations in the browser such as Microsoft PowerPoint®.
- ▶ Storage: Storage of offline data on the client side. Implementation provides functionality to store data with Flash or Google Gears.
- ▶ String: Extended string functionality including a formatter, tokenizer, or string builder.
- ▶ Timing: Advanced use of timing constructs.

- ▶ **Uuid:** Provides functionality to generate universally unique identifiers that are unique across space and time.
- ▶ **Validate:** Validation functions for select formats such as e-mail addresses or credit card numbers.
- ▶ **Widget:** Additional widgets such as Toaster, FileInput, FishEye, or TimeSpinner.
- ▶ **Wire:** Provides functionality to declaratively link widgets, HTML objects, and so on, together and link up data from different sources such as XML DOM elements, dojo.data data items, and so on, to different targets.
- ▶ **Xml:** A set of tools for simplifying XML-data handling.

## 9.11 Unit testing with the Dojo Objective Harness

The doh folder under the util folder contains the Dojo Objective Harness (DOH). DOH is a testing framework for JavaScript applications. It provides a browser-based test-runner. Tests can also be run via the command line together with Rhino, a JavaScript implementation written in Java. Dojo itself provides numerous DOH tests for its own modules. Dojo core tests are placed in dojo/tests and Dijit tests in dijit/tests. Each DojoX project contains its own tests folder.

Command-line testing with Rhino is not covered in this chapter. Instructions can be found at:

<http://dojotoolkit.org/book/dojo-book-0-9/part-4-meta-dojo/d-o-h-unit-testing>

### 9.11.1 Writing tests

This section describes how you can test the functions, classes, and widgets that use the Dojo library functions.

## Folder structure

Figure 9-23 shows how the testing folder structure might look.

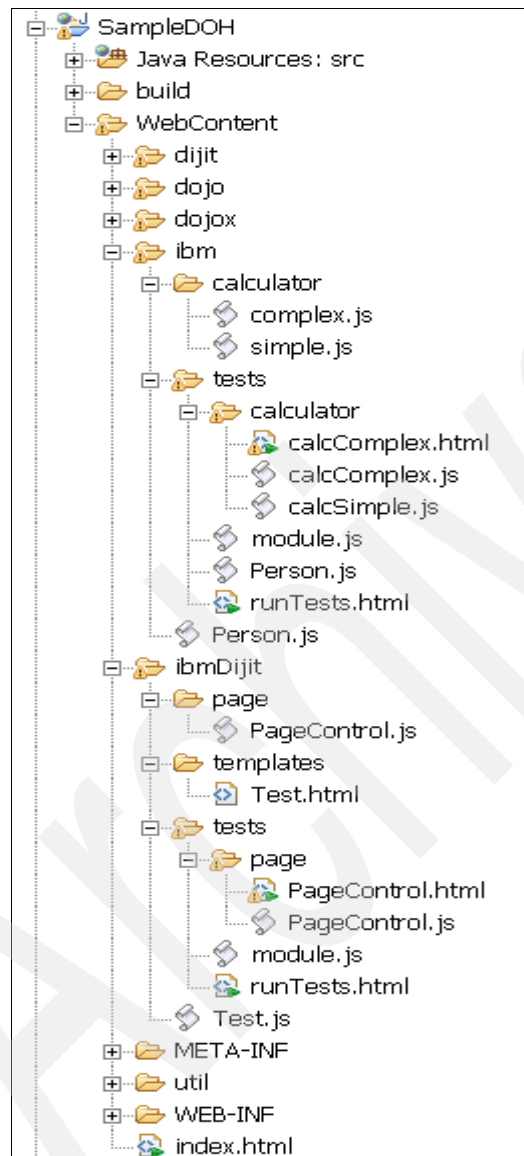


Figure 9-23 Testing folder structure for Dojo's Objective Harness

Each package to be tested should contain a tests folder. In this example these are the `ibm` and `ibmDijit` folders. Inside the tests folder is the same subfolder structure as the package being tested.

## runTests HTML

Create a runTests.html in the tests folder. Example 9-96 shows the contents of the runTests.html in the ibm/tests folder.

*Example 9-96 Contents of runTests.html*

---

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>IBM Functions and Class Test Runner</title>
<meta http-equiv="refresh"
content="0;url=../../util/doh/runner.html?testModule=ibm.tests.module">
</head>
<body>
    Redirecting to D.O.H runner.
</body>
</html>
```

---

This HTML page redirects to the runner.html that is provided by the DOH. It runs and displays the test results as shown in 9.11.2, “Running tests” on page 397.

## Module loader

In order to run a test you must pass a testModule parameter to the runner.html file (see the parameter list in the example above). It contains all of the JavaScript files being tested. Example 9-97 shows the content of the module.js file in ibm/tests.

*Example 9-97 Content of the tests module file*

---

```
dojo.provide("ibm.tests.module");

dojo.registerModulePath("tests", "../../ibm/tests");

try {
    dojo.require("tests.calculator.calcComplex");
    dojo.require("tests.calculator.calcSimple");
    dojo.require("tests.Person");
} catch(e){
    doh.debug(e);
}
```

---

## Assertions

When using DOH you have access to the global variables `doh` and `tests`. The `tests` variable is just an alias for the `doh` variable. The `doh` variable has the following test assertion functions:

- ▶ `doh.assertEqual` or `doh.is` (for example, `doh.is(expectedVal, actualVal)`)
- ▶ `doh.assertError` or `doh.e` (for example, `doh.e(errorObject)`)
- ▶ `doh.assertTrue` or `doh.t` (for example, `doh.assertTrue(condition)`)
- ▶ `doh.assertFalse` or `doh.f` (for example, `doh.f(condition)`)

## Format of test functions

Tests can be registered with the `doh.register` function. A test function can have one of the following formats:

- ▶ Normal function: It is good practice to start your function name with the string `test`, although it is not mandatory. The function name is displayed in the test results (Example 9-98).

*Example 9-98 DOH test as normal function*

---

```
function testFunc(t){  
    // assert call  
    t.t(condition);  
}
```

---

- ▶ Object: If you pass the function in the form of an object you can define a `setUp` and `tearDown` method. The `name` property is displayed in the test results (Example 9-99).

*Example 9-99 DPH test passed as an object*

---

```
{  
    name: "testFunc",  
    setUp: function() {  
        // is called before the test runs  
    },  
    runTest: function(t) {  
        doh.assertTrue(someCondition);  
    },  
    tearDown: function() {  
        // do some cleanup if needed  
    }  
}
```

---

- ▶ String: You can pass the function in the form of a string like "doh.is(someObject.addNums(10,10), 20)" and Dojo evaluates it.
- ▶ Asynchronous function: To run a test asynchronously, your test function must return a doh.Deferred object in order to be handled as asynchronous.

Example 9-100 shows how you can write a test function that uses setTimeout.

*Example 9-100 DOH test for asynchronous functions*

---

```
function testAsynchFunc(t){
    var d = new doh.Deferred();

    setTimeout(d.getTestCallback(function() {
        t.t(condition);
    }), 100);
    return d;
}
```

---

Example 9-101 shows how you write test functions that make a server request.

*Example 9-101 DOH test for testing asynchronous communication*

---

```
function testAsynchFunc(t){
    var d = new doh.Deferred();
    var xhr = dojo.xhrGet({
        url: "/server/servlet", // self
        content: { key: "val" }
    });
    xhr.addCallback(function(resp){
        t.is(expectedString, resp);
        d.callback(true);
    });
    return d;
}
```

---

The following section shows how to use these function formats in an actual JavaScript test file.

## Test pure JavaScript

The content of the tests.calculator.calcSimple file might look like Example 9-102.

*Example 9-102 Content of tests.calculator.calcSimple*

---

```
dojo.provide("tests.calculator.calcSimple");
dojo.require("ibm.calculator.simple");

tests.register("tests.calculator.calcSimple",
[
    function testAdd(t){
        tests.assertEqual(30, ibm.calculator.simple.add(10, 20));
    },
    function testSubtract(t){
        t.is(40, ibm.calculator.simple.subtract(10, -30));
    },
    "doh.is(20, ibm.calculator.simple.add(10, 10))",
    {
        name: "testIsNumber",
        setUp: function(){
            // do something
        },
        runTest: function(t){
            t.t(ibm.calculator.simple.isNumber(5));
        },
        tearDown: function() {
            // cleanup
        }
    },
    function testIsNumberAsynch(t){
        var d = new doh.Deferred();
        setTimeout(d.getTestCallback(function() {
            t.f(ibm.calculator.simple.isNumber("string"));
        }), 100);
        return d;
    }
]
)
```

---

## Test together with an HTML document

The previous example shows how to test pure JavaScript functionality. If you want to test modules together with an HTML document like the testing of widget functionality, the module.js does not contain the registration of the tests. Instead,

it contains a reference to the HTML file that is used for testing. Example 9-103 shows the content for the `ibmDijit/tests/page/PageControl.js` file.

*Example 9-103 Content of tests.page.PageControl*

---

```
dojo.provide("tests.page.PageControl");

if(dojo.isBrowser){
    doh.registerUrl("tests.page.PageControl",
        dojo.moduleUrl("tests.page", "PageControl.html"), 2000);
    // additional dojo code
}
```

---

The property `dojo.isBrowser` determines whether the tests are being run utilizing a browser. Example 9-104 shows the content of the `ibmDijit/tests/page/PageControl.html`.

*Example 9-104 Content of PageControl.html*

---

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
    "http://www.w3.org/TR/html4/strict.dtd">
<html>
<head>
<title>Testing Calculator Complex</title>
<script type="text/javascript" src="../../dojo/dojo.js"
    djConfig="isDebug: true, parseOnLoad: true"></script>
<script type="text/javascript">
    dojo.require("doh.runner");
    dojo.require("ibmDijit.page.PageControl");

    dojo.addOnLoad(function(){
        doh.register("t",
            [
                function testPageWidth(t){
                    t.is("500px", dijit.byId("pageControl").getWidth());
                },

                function testAddPage(t){
                    var pageCtrl = dijit.byId("pageControl");
                    var page = dojo.doc.createElement("div");
                    pageCtrl.addPage(page);
                    t.is(1, pageCtrl.getTotalPageNum());
                }
            ]
        );
        doh.run();
    });
```





## 9.12 IBM Extensions to the Dojo Toolkit

The Ajax client runtime contains additional IBM client-side JavaScript libraries to simplify working with different protocols, data formats, and connectivity options.

The IBM Extensions to the Dojo Toolkit are:

- ▶ IBM Atom library

The Atom library, data store, and widgets comprise the IBM Atom library. The Atom Publishing Protocol (APP) data access makes it easy to invoke any Atom APP 1.0-compliant service, and uses Atom feeds as data sources that are bound to widgets with an Ajax application.

Atom feed widgets are sample feed widgets that use the Atom APP data access feature.

- ▶ IBM Gauge widgets

Two widgets are supplied: the Analog Gauge widget and the bar Gauge widget. These widgets are useful for displaying monitored values, and they support real-time updates.

- ▶ The OpenSearch library

This library makes it easy to invoke any OpenSearch-compliant service and bind search results to widgets within an Ajax application.

- ▶ IBM SOAP library

SOAP connectivity makes it easier to invoke public SOAP-based Web services from Ajax applications.

IBM Extensions to the Dojo Toolkit require Dojo 1.0 or later.

### 9.12.1 IBM Atom library

The IBM Atom library is comprised of three parts:

- ▶ The Atom library

This library contains three features:

- The general utility functions to support the rest of the library.
- The data model for the various parts of Atom, such as content, person, link, feed, and entry. These data models are then used to define the AtomIO object.
- The AtomIO object is a wrapper to the various functions intrinsic to Atom feeds and the Atom Publishing Protocol (APP).

► The AppStore

This data store implements the dojo.data Read API, the Identity API, and the Write API. The AppStore handles reading and writing from an APP source in an implementation agnostic way. It also supports fetching and storing entries without knowledge of the APP underpinnings.

► The Atom widgets

Included in the package are three widgets:

- FeedViewer: The FeedViewer is used for displaying the title and dates of entries in a feed.
- FeedEntryViewer: The FeedEntryViewer is used for displaying the details of a selected Atom Entry in a FeedViewer.
- FeedEntryEditor, similar to FeedEntryViewer, but allows for editing of existing entries and creation of new ones.

Due to the same-origin policy present in browsers, JavaScript code cannot request resources from servers other than the server from which the current script originated.

There are several workarounds, however. The Atom library uses the standard XMLHttpRequest. Because of this, the library can only handle feeds that originate from the same domain that the library is served from.

To retrieve feeds from other domains, use the Ajax Proxy (see 6.3, “Ajax Proxy” on page 131) to broker requests to external resources.

## Package structure

The IBM Atom library, data store, and widgets are distributed in two different packages:

- IBM WebSphere Application Server Feature Pack for Web 2.0 installer places the package structure outlined below into the folder located at `<app_server_root>/Web2fep/ajax-rt_1.X/ibm_atom`.
- The Stand-alone package includes the given package structure inside the folder located at `/AjaxClientRuntime_1.X/ibm_atom`.

The package is organized into the structure shown in Figure 9-25.

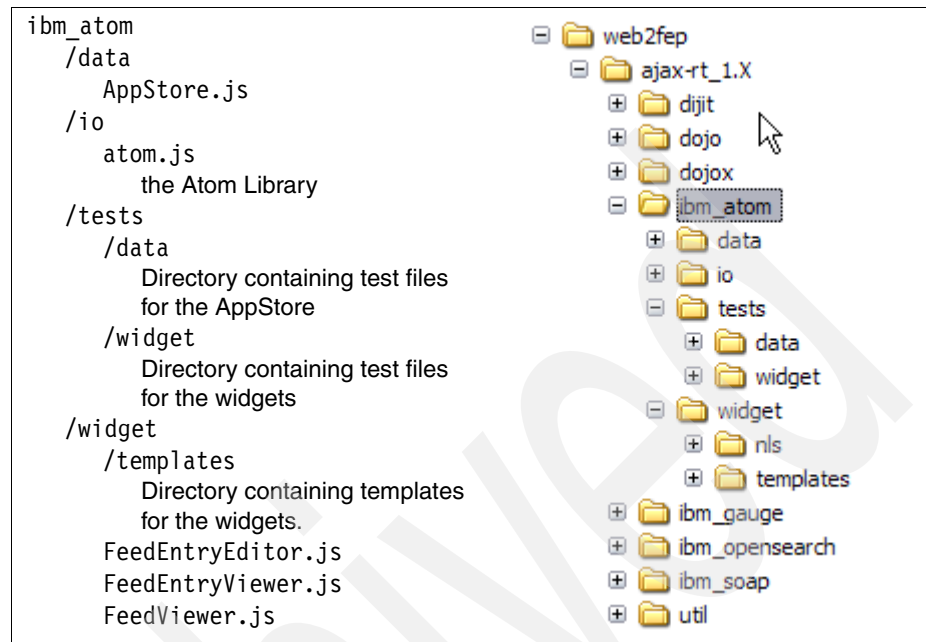


Figure 9-25 IBM Atom library package structure

## The Atom library

The Atom library provides a base set of functionality for using Atom feeds. It provides a data model to handle accessing the various elements of Atom feeds and entries, as well as a transport object to handle the various interactions with Web servers and for handling both the standard Atom feeds and the Atom Publishing Protocol.

The Atom Syndication Specification is defined at:

<http://www.atomenabled.org/developers/syndication/atom-format-spec.php>

The Atom Publishing Protocol is available at:

<http://bitworking.org/projects/atom/>

## Getting a feed

Retrieving a feed is simple. With all Atom server interactions with a client, an AtomIO object is created. This is the transport object referred to above, which handles all interactions with the Atom server. We will start with a simple task, such as retrieving a feed. This is the sample feed being retrieved (samplefeed.xml):

```
<?xml version='1.0' encoding='utf-8'?>
<feed xmlns='http://www.w3.org/2005/Atom' xml:lang='en-US'>
  <title>Example.com</title>
  <link rel="alternate" type="text/html" href="http://example.com/"
hreflang="en" title="Example.com" />
  <subtitle type='text'>Example.com's Sample Feed</subtitle>
  <rights>Copyright Example.com</rights>
  <id>http://example.com/samplefeedEdit.xml</id>
  <updated>2007-08-07T20:00:00-05:00</updated>
  <link rel="self" type="application/atom+xml"
href="http://www.example.com/samplefeed.xml"/>
  <entry>
    <title>Test Entry #1</title>
    <id>http://example.com/samplefeed.xml/entry/1</id>
    <link rel='alternate' href='http://example.com/1.html' />
    <summary type='html'>
      <p>This is a sample entry in our Atom feed.</p>
    </summary>
    <author>
      <name>Test User</name>
      <email>test@example.com</email>
    </author>
    <updated>2007-08-07T04:00:00-05:00</updated>
  </entry>
  <entry>
    <title>Test Editable Entry #1</title>
    <id>http://example.com/samplefeed.xml/entry/2</id>
    <link rel='alternate' href='http://example.com/2.html' />
    <link rel='edit'
href='http://example.com/samplefeed.xml/entry/edit/2' />
    <summary type='text/html'>
      <p>This is another sample entry in our Atom feed except this
entry is also editable!</p>
    </summary>
    <author>
      <name>Test User</name>
      <email>test@example.com</email>
    </author>
```

```
        <updated>2007-08-07T06:00:00-05:00</updated>
    </entry>
</feed>
```

To create a feed object from this feed, use the code shown in Example 9-105.

---

*Example 9-105 Creating a feed object from the sample feed*

---

```
var _feed = null;
var atomIO = new ibm_atom.io.atom.AtomIO();

function getComplete(feed, domNode){
    _feed = feed;
}

atomIO.getFeed("samplefeed.xml", getComplete);
```

---

This block of code first defines a `_feed` variable, creates an `AtomIO` object, defines a function to be used as a callback, and then finally calls the `getFeed` function on the `AtomIO` object to fetch the feed. In the previous code block, the string "samplefeed.xml" is the URL to the feed.

### ***Getting a feed entry***

With a feed object, you can fetch any of the attributes of the feed (such as the title, the ID, authors, contributors, updated date, and so on). Additionally, the entries beneath the feed can be fetched, either as an array of entries or individually by ID. See Example 9-106.

---

*Example 9-106 Working with the feed object*

---

```
function getComplete(feed, domNode){
    _feed = feed;

    // id now equals "http://www.onlamp.com/"
    var id = _feed.id;

    // _feed.title is a ibm_atom.io.atom.Content object, with
    // a value of "Example.com"
    var title = _feed.title.value;

    // an array of all entries. Current length is 3.
    var arrayOfEntries = _feed.entries;

    // the entry specified by the given ID string. In this case,
    // it is the first entry in our feed.
    var specificEntry =
```

```
    _feed.getEntry("http://example.com/samplefeed.xml/entry/2");  
}
```

---

With an individual entry object, you can perform similar functions on a feed object, including getting the entry title, ID, updated date, and so on. The only unique function available with an entry object is the ability to get the edit URL for that entry. This ability is not shown in Example 9-107, as the sample feed does not include any editable entries.

*Example 9-107 Working with eEntry objects*

---

```
function getComplete(feed, domNode){  
    _feed = feed;  
  
    var specificEntry =  
        _feed.getEntry("http://example.com/samplefeed.xml/entry/2");  
  
    // the id of the entry.  
    var id = specificEntry.id;  
  
    // the entry's title is an ibm_atom.io.atom.Content object,  
    // the text value is in the 'value' attribute of that object.  
    var title = specificEntry.title.value;  
  
    // the entry's updated Date object.  
    var updated = specificEntry.updated;  
}
```

---

The other data model objects, such as authors and contributors, links, and so on, all operate similarly and have their own attributes and functions. For more information, check out the Atom reference contained in the Atom library, accessible through the InfoCenter for Feature Pack for Web 2.0.<sup>1</sup>

---

<sup>1</sup> <http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.dojo-atom.help/docs/Reference.html>

### ***The AtomIO object***

The AtomIO object supports functions for sending data back to the Atom server, including getting a single entry by use of the `getEntry` function. This is typically to parse an Atom entry document, but if presented with a feed document, it returns the first entry of that feed. See Example 9-108.

#### *Example 9-108 Working with the AtomIO object*

---

```
var _entry = null;
var atomIO = new ibm_atom.io.atom.AtomIO();

function getComplete(/* Atom Entry Object */ entry,
    /* DOM Node Object */ domNode) {
    _entry = entry;

    // the id of the entry.
    var id = _entry.id;

    // the entry's title is an ibm_atom.io.atom.Content object,
    // the text value is in the 'value' attribute of that object.
    var title = _entry.title.value;

    // the entry's updated Date object.
    var updated = _entry.updated;
}

atomIO.getEntry("samplefeed.xml", getComplete);
```

---

You can also use the AtomIO object to send updated entries back to the Atom server, with the URL being fetched from the edited entry. See Example 9-109.

#### *Example 9-109 Sending updated entries back to the server*

---

```
var _entry = null;
var atomIO = new ibm_atom.io.atom.AtomIO();

function getComplete(entry, domNode){
    _entry = entry;

    // change editURL to a valid URL
    _entry.addLink("editURL", "edit");

    _entry.title.value = "Test Title!";

    function getComplete2(entry2, domNode){
        // ...
    }
}
```



```

    }

    // an APP server is required to respond to this request
    atomIO.updateEntry(_entry, getComplete2);
}

atomIO.getEntry("samplefeedEdit.xml", getComplete);

```

---

The Atom reference<sup>2</sup> outlines more AtomIO functions available.

## The AppStore

The AppStore data store can be used just as any other data store that conforms to the dojo.data APIs. However, the AppStore data store does require a working APP feed for the write functions to work. The application shown in Example 9-110, uses the same samplefeed.xml file as with the Atom library examples, and demonstrates the Read and Identity API functions of the AppStore data store.

An AppStore application is created in markup, declaratively, from the feed at the given URL.

*Example 9-110 Creating an AppStore application declaratively*

---

```

<head>
...
<script type="text/javascript">djConfig = {parseOnLoad: true};</script>
<script type="text/javascript" src="../../dojo/dojo.js"></script>
<script>
    dojo.require("dojo.parser");
    dojo.require("ibm_atom.data.AppStore");
</script>
...
</head>
...
<div dojoType="ibm_atom.data.AppStore" url="samplefeed.xml"
jsId="appStore">

```

---

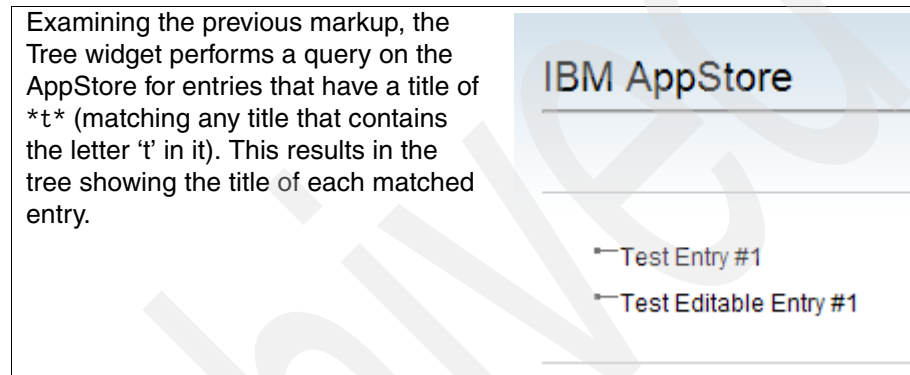
<sup>2</sup> <http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.dojo-atom.help/docs/Reference.html>

Now the store has been instantiated and the feed is loaded. As an example of some of the more advanced tasks that can be undertaken with a store, you can add in a tree widget, which gets its data from the AppStore and display a list of titles. See Example 9-111.

*Example 9-111 Dojo's tree using IBM AppStore*

```
<div dojoType="dijit.Tree" id="tree" store="appStore"
query="{title: '*t*'}"></div>
```

The Tree widget with the display of titles is shown in Figure 9-26.



*Figure 9-26 Dojo's tree using IBM AppStore*

Let us examine some simple functions of the AppStore, the first of which is fetching items from the feed. Suppose that you want to fetch the entry with the title "Test Entry #1". You might chose to run a fetch against the AppStore, as shown in Example 9-112.

*Example 9-112 Fetching the entry with the title "Test Entry #1"*

```
<script type="text/javascript">
dojo.addOnLoad(function(){
    function callback(items, request){
        console.debug(items.length);
        if(items.length > 0)
            console.debug(appStore.getValue(items[0], "title");
    }
    appStore.fetch({
        query: {title: "Test Entry #1"},
        onComplete: callback});
});
</script>
```

Having now fetched an item, there are several functions that you can call with it. This includes getting attribute values of the item, checking whether the item is an item of our store, and examining the identity of the item. See Example 9-113.

*Example 9-113 Working with dojo.data items from the AppStore*

---

```
<script type="text/javascript">
dojo.addOnLoad(function() {
    function callback(items, request){
        var item = items[0];
        console.debug(appStore.getValue(item, "summary"));
        console.debug(appStore.isItem(item));
        console.debug(appStore.getIdentity(item));
    }
    appStore.fetch({
        query: {title: "Test Entry #1"},
        onComplete: callback});
});
</script>
```

---

See 9.9, “Data API” on page 371, for details of further Dojo Data API functionality.

## The Atom widgets

Three top-level widgets are included in the `ibm_atom` package:

- ▶ `FeedViewer`
- ▶ `FeedEntryViewer`
- ▶ `FeedEntryEditor`

`FeedViewer` is used to view the dates and titles of entries in a feed. It can be instantiated declaratively in markup or programmatically. Example 9-114 demonstrates creating a feed viewer declaratively.

*Example 9-114 Creating a FeedViewer declaratively*

---

```
<div dojoType="ibm_atom.widget.FeedViewer"
    widgetId="fv1"
    url="samplefeed.xml"
    entrySelectionTopic="atomfeed/entry/topic" />
```

---

The `url` value specifies the address from which to retrieve the Atom feed (be aware of the browser’s same origin policy here; a proxy server may be required). The `entrySelectionTopic` is the topic to both subscribe and publish. This enables two-way communication between the `FeedViewer` and `FeedEntryViewer` or

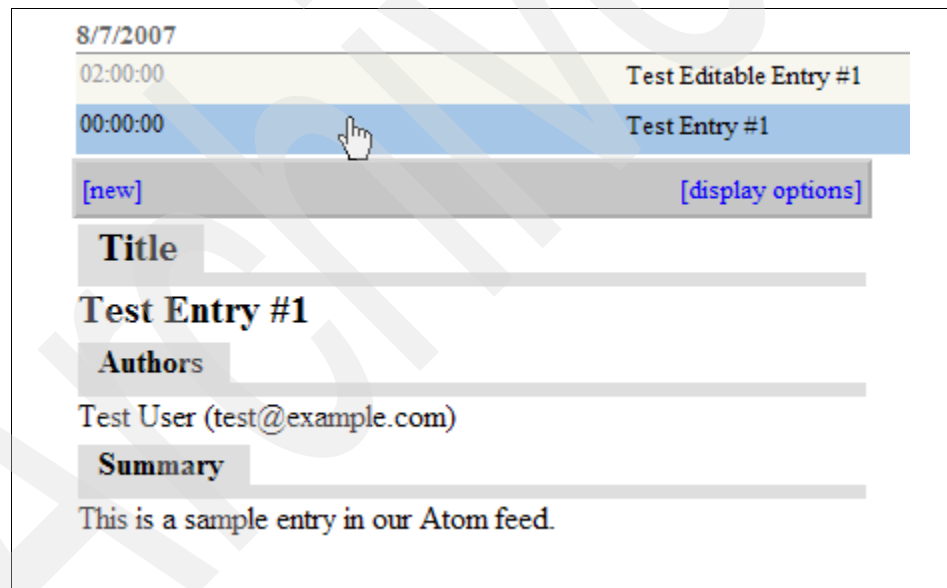
FeedEntryEditor widgets. Note that if the code above is put into a test page, the example is completely unstyled and simple.

The FeedEntryViewer widget displays an individual entry element. A bar is provided at the top with a menu to turn specific elements on or off. The viewable elements can be dictated at creation time. See Example 9-115.

*Example 9-115 Creating a FeedEntryViewer declaratively*

```
<div dojoType="ibm_atom.widget.FeedEntryViewer"
    widgetId="feedEditor"
    enableMenu="true"
    enableMenuFade="false"
    displayEntrySections="title,authors,summary"
    entrySelectionTopic="atomfeed/entry/topic" />
```

Figure 9-27 shows the output using both the FeedViewer and FeedEntryViewer widgets.



*Figure 9-27 FeedViewer widget above the FeedEntryViewer widget*

The FeedEntryEditor is very similar to the FeedEntryViewer except it permits the feed entries to be edited by setting the enableEdit attribute to true, and if the entry has a link with a rel attribute that equals to "edit" (as in Test Editable Entry #1).

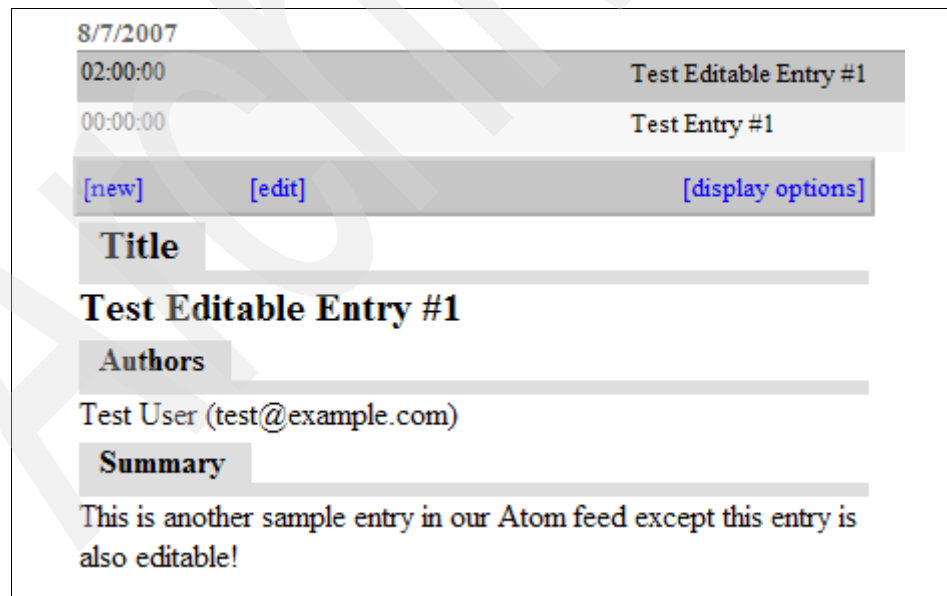
This widget does none of the work of sending the edits back as requests. Instead, the widget updates its rendering and sends the updates on the entrySelectionTopic there by allowing the FeedViewer widget to handle the sending of the data.

The FeedEntryEditor widget extends the FeedEntryViewer widget. Therefore, all of its variables and functions are inherited. This widget does, however, override the previously mentioned undocumented functions that set the display node values, as it must replace the value with an editor object in the event that this entry is being edited. See Example 9-116.

*Example 9-116 Creating a FeedEntryEditor widget declaratively*

```
<div dojoType="ibm_atom.widget.FeedEntryEditor"
    widgetId="feedEditor"
    enableMenu="true"
    enableMenuFade="true"
    enableEdit="true"
    displayEntrySections="title,authors,summary"
    entrySelectionTopic="atomfeed/entry/topic" />
```

Figure 9-28 shows the output using both the FeedViewer and FeedEntryEditor widgets.



*Figure 9-28 FeedViewer widget above the FeedEntryEditor widget: note [edit] on the bar*

The entrySelectionTopic is common to the FeedViewer, FeedEntryViewer, and FeedEntryEditor. This is the event that ties them together. If the entrySelectionTopic attribute is defined properly on each, then the two widgets can communicate between each other and update themselves appropriately.

The Atom reference<sup>3</sup> details the variables and functions available for Atom widgets.

## 9.12.2 IBM Gauge widgets

Gauge widgets are built on a standard base \_Gauge object. This object can be used as a base object for developing Gauge widgets. The base object is not to be used as a standalone widget.

The Gauge widget library includes a pair of widgets for displaying numerical data in a graphically rich way. Using Scalable Vector Graphics (SVG) or Vector Markup Language (VML), depending on the browser, the AnalogGauge and BarGauge widgets display numerical data with customizable ranges, tick marks, and indicators. The Gauge widgets can be used to create dynamically self-updating graphical displays and dashboards.

### Package structure

The Gauge widgets are distributed in two different packages:

- ▶ The IBM WebSphere Application Server Feature Pack for Web 2.0 installer places the package structure outlined below into the folder located at <app\_server\_root>/Web2fep/ajax-rt\_1.X/ibm\_gauge.
- ▶ The Stand-alone package includes the given package structure inside the folder located at /AjaxClientRuntime\_1.X/ibm\_gauge.

---

<sup>3</sup> <http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/topic/com.ibm.dojo-atom.help/docs/Reference.html>

The Gauge widget package is organized in the structure shown in Figure 9-29.

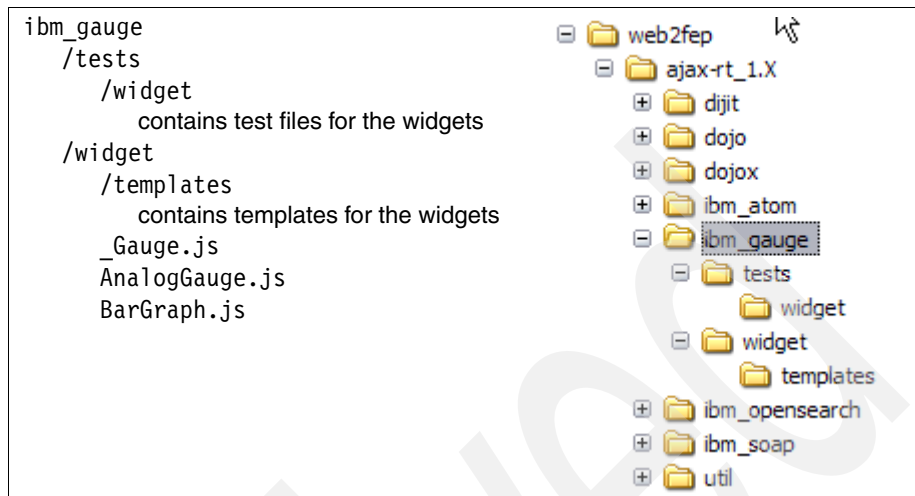


Figure 9-29 The Gauge widget library package structure

### Analog Gauge widget

The Analog Gauge widget provides a means for displaying data on a circular gauge, such as a speedometer or a pressure gauge. To create an analog gauge, first determine the shape that you want the gauge to take, what indicators you want to display on the gauge, what ranges you want to create on the range, and other factors. See Figure 9-30.

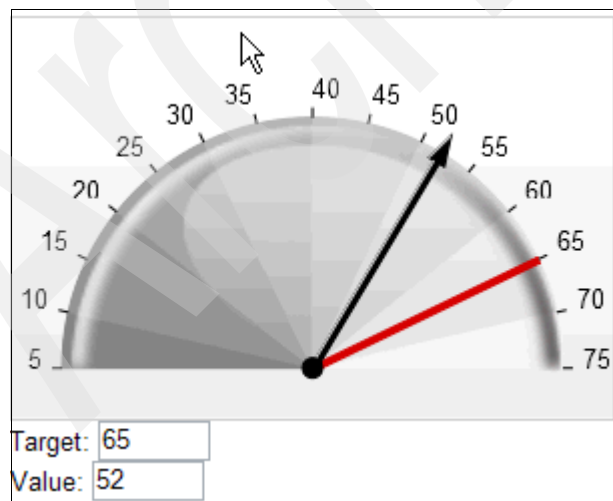


Figure 9-30 The Analog Gauge widget

The code given in Example 9-117 creates a new Analog Gauge widget, located within the div with an ID of testGauge. The gauge currently has no ranges, indicators, or tick marks. The data area for the gauge is defined by the parameters passed in Example 9-117.

*Example 9-117 Creating an analog gauge*

---

```
<div dojoType="ibm_gauge.widget.AnalogGauge"
    id="testGauge"
    cx=150
    cy=175
    radius=125>
</div>
```

---

The code shown in Example 9-118 adds five ranges to the gauge, with appropriate low, high, and hover values. The hover value is displayed when the mouse points to that particular range.

*Example 9-118 Adding ranges to the gauge*

---

```
var g = dijit.byId("testGauge");
g.addRange({low:0, high:10, hover:'0 - 10'});
g.addRange({low:10, high:20, hover:'10 - 20'});
g.addRange({low:20, high:30, hover:'20 - 30'});
g.addRange({low:30, high:40, hover:'30 - 40'});
g.addRange({low:40, high:50, hover:'40 - 50'});
```

---

We now want to add tick marks at every five units, with a small line and the value indicated in text. Using a *for* loop is a more efficient process than writing out the values for each individual tick mark. See Example 9-119.

*Example 9-119 Adding tick marks to the gauge*

---

```
for(var i=0; i<=50; i+=5){
    g.addIndicator({ value:i,
        type:'line',
        length:5,
        offset:125,
        label:''+i
    });
}
```

---

Next we add an arrow indicator to the gauge. The value parameter specifies the current numerical value on the gauge where the indicator is positioned initially. The type indicates the type of indicator. For an analog gauge, the available values are arrow and line. The length and width indicate the size of the arrow,



and the hover and title parameters specify the text attributed with the indicator. See Example 9-120.

*Example 9-120 Adding value indicator line to the gauge*

---

```
var valueIndicator = {
  value:17,
  type:'arrow',
  color: 'yellow',
  length: 135,
  width: 3,
  hover:'Value: 17',
  title: 'Value'
};
g.addIndicator(valueIndicator);
```

---

Now that we have created a target indicator, we also want to indicate a goal or a warning threshold. You can add any number of indicators to the gauge, as shown in Example 9-121.

*Example 9-121 Adding target indicator line to the gauge*

---

```
var targetIndicator = {
  value:6,
  type:'line',
  color:'red',
  width: 3,
  hover:'Target: 6',
  title: 'Target'
};
g.addIndicator(targetIndicator);
```

---

The indicators with titles have boxes beneath the gauge that indicate numerical values. These boxes are displayed by default to ensure that this widget is accessible to those using screen readers to view the page. The boxes can be disabled by setting the `hideValues` parameter to true on the gauge itself. However, disabling the boxes does make the widget inaccessible for those using screen readers. The `hideValues` parameter functions the same on the bar graph widget as well.

## Bar Graph widget

The bar graph widget is used to display numerical information graphically in a bar graph. As with the Analog Gauge widget, multiple possibilities exist. You can add any number of bars, indicators, tick marks, text, and so on to the widget. See Figure 9-31.

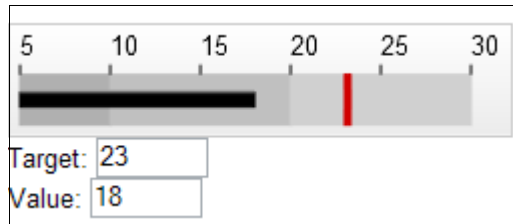


Figure 9-31 The Bar Graph widget

The code shown in Example 9-122 creates a new bar graph Gauge widget, located within the div with an ID of testBarGraph. The gauge currently has no ranges, indicators, or tick marks. The data area for the bar graph is specified by the values assigned to the gaugeHeight, dataY, dataHeight, and dataWidth variables.

Example 9-122 Creating a bar graph gauge

```
<div dojoType="ibm_gauge.widget.BarGraph"
  id="testBarGraph"
  widgetId="testBarGraph"
  executeScripts="true"
  gaugeHeight="55"
  dataY="25"
  dataHeight="25"
  dataWidth="225">
</div>
```

You can add a gradient background to the bar graph as shown in Example 9-123. This code adds a vertical, linear gradient, going from a white color at the beginning to a light gray at the end, for the background of the bar graph. Three types of gradients are available. The same applies to the Analog Gauge widget as well, as they are both handled by the `_Gauge` super class.

Example 9-123 Adding gradient background to the bar graph

```
var bg = dijit.byId("testBarGraph");
var fill = {
  type: "linear",
  x1: bg.gaugeWidth/2,
```

```

    x2: bg.gaugeWidth/2,
    y2: 0,
    y1: bg.gaugeHeight,
    colors: [{offset: 0, color: "#ECECEC"}, {offset: 1, color: "white"}]
  };
  bg.gaugeBackground = fill;

```

---

Just as with the Analog Gauge widget, you can add the ranges and tick marks shown in Example 9-124.

*Example 9-124 Adding ranges and tick marks to the bar graph*

---

```

bg.addRange({low:0, high:10, hover:'0 - 10'});
bg.addRange({low:10, high:20, hover:'10 - 20'});
bg.addRange({low:20, high:30, hover:'20 - 30'});
bg.addRange({low:30, high:40, hover:'30 - 40'});
bg.addRange({low:40, high:50, hover:'40 - 50'});

for(var i=0; i<=50; i+=5){
  bg.addIndicator({
    value:i,
    type:'line',
    length:5,
    offset:-5,
    width: 1,
    label:''+i
  });
}

```

---

Finally, you can add a target and value indicators. For the bar graph widget, the target indicator is still the *line* type. However, the value indicator is the *bar* type, as shown in Example 9-125.

*Example 9-125 Adding target and value indicators to the bar graph*

---

```

var valueIndicator = {
  value:17,
  type:"bar",
  width: 7,
  hover:'Value: 17',
  title: 'Value'
}
bg.addIndicator(valueIndicator);

var targetIndicator = {

```

```
value:22,  
type:"line",  
color:'#D00000',  
hover:'Target: 22',  
title: 'Target'  
}  
bg.addIndicator(targetIndicator);
```

---

The bar graph is now complete. The target and value indicators can be moved with a drag action from the graph to new values. In addition, you can change the values in the value boxes beneath the bar graph. As with the analog gauge, those boxes that display the values can be disabled.

### 9.12.3 IBM OpenSearch library

The OpenSearch library makes it easy to invoke any OpenSearch-compliant service and bind search results to widgets within your Ajax application.

The library is intended to interface with a server with open search capabilities. Typically, server-side support consists of hosting an open search description document that defines the URLs used to query the server. The server can return the data in any number of formats, with the return type defined in the URL element within the description document. Typically, the return type is either X/HTML, an Atom, or RSS feed.

The OpenSearch library includes a data store conforming to the `dojo.data Read API` (see 9.9.3, “Read API” on page 377). This data store is instantiated using a URL to a open search description document. Unless you have a proxy server set up, such as the `AjaxProxy` component (see 6.3, “Ajax Proxy” on page 131), this document and all search endpoints must be on the same server (that is, the same domain) as the page that the data store is created in, due to the cross-domain limitations of browsers.

When the store is instantiated, it first parses the description document, then determines the best URL element to use and, if there are multiple elements, selects them based on the ease of parsing the results. The types, in order, are Atom, RSS, and HTML. When the results are fetched from the data store using a query, it replaces the `searchTerms` parameter and any other supplied parameters, based on the template and what is provided in the request. The store queries the server to retrieve the results. Again, due to cross-domain browser limitations, the URL endpoints must be on the same domain as the page on which the datastore resides, unless a proxy server is being used.

## Package structure

The OpenSearch data store is distributed in two different packages:

- ▶ The IBM WebSphere Application Server Feature Pack for Web 2.0 installer places the package structure outlined below into the folder located at `<app_server_root>/Web2fep/ajax-rt_1.X/ibm_opensearch`.
- ▶ The stand-alone package includes the given package structure inside the folder located at `/AjaxClientRuntime_1.X/ibm_opensearch`.

The package is organized into the structure shown in Figure 9-32.

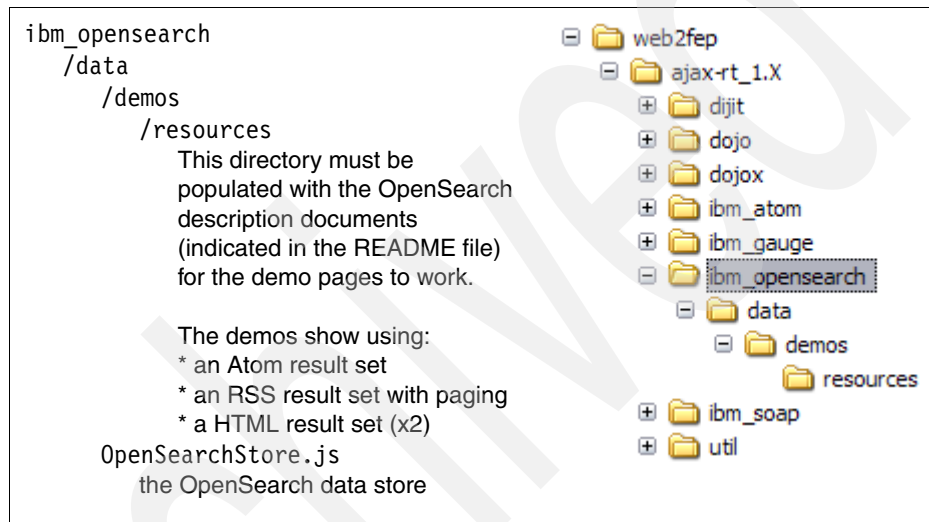


Figure 9-32 The IBM OpenSearch library package structure

## The OpenSearch data store

Since cross-domain requests are not possible in current browsers unless the OpenSearch description documents are present on the same server as the OpenSearch data store, a proxy server (like the Ajax Proxy contained in this feature pack) must be used so that the requests to send and fetch the documents go through the proxy.

The OpenSearch data store is an implementation of the `dojo.data` Read API and as such may be created declaratively or programmatically.

Example 9-126 shows the creation of an OpenSearch data store declaratively.

*Example 9-126 Creating OpenSearch data store declaratively*

---

```
<script src="dojo/dojo.js"></script>
<script>
    dojo.require("ibm_opensearch.data.OpenSearchStore");
    dojo.require("dojo.parser");
</script>
<div dojoType="ibm_opensearch.data.OpenSearchStore"
    url="osd.xml"
    jsId="openSearchStore">
</div>
```

---

Example 9-127 shows the creation of an OpenSearch data store programmatically.

*Example 9-127 Creating OpenSearch data store programmatically*

---

```
dojo.require("ibm_opensearch.data.OpenSearchStore");
...
var openSearchStore =
    new ibm_opensearch.data.OpenSearchStore({url:"osd.xml"});
```

---

The previous two examples assume that the data store is being instantiated within the page `http://example.com/index.html`.

The following is an example of an OpenSearch description document (hosted at `http://example.com/osd.xml` with a mime-type of `application/opensearchdescription+xml`) with only the required elements included:

```
<OpenSearchDescription xmlns="http://a9.com/-/spec/opensearch/1.1/">
  <ShortName>Sample Search Engine</ShortName>
  <Description>Searches each element at example.com</Description>
  <Url type="application/atom+xml"
    template="http://example.com/search/{searchTerms}" />
</OpenSearchDescription>
```

You can use the store to fetch results based on supplied keywords, as shown in Example 9-128.

*Example 9-128 Fetching results using OpenSearch data store*

---

```
<table>
  <tbody id="searchResults">
  </tbody>
```

```

</table>
<script>
dojo.addOnLoad(function(){
    function onComplete(items, request){
        if(items.length > 0){
            var tbody = dojo.byId("searchResults");
            var test;
            var tr, td;
            for(var i=0; i<items.length; i++){
                td = dojo.doc.createElement("td");
                td.innerHTML =
                    openSearchStore.getValue(items[i], "content");
                tr = dojo.doc.createElement("tr");
                tr.appendChild(td);
                tbody.appendChild(tr);
            }
        }
    }
    //What to do if a search fails
    function onError(error, request){
        statusWidget.setValue("PROCESSING ERROR.");
    }
    var request = {
        query: {searchTerms: "atom"},
        onComplete: onComplete,
        onError: onError
    };
    openSearchStore.fetch(request);
});
</script>

```

---

The request object sets the searchTerms parameter of the query to atom (the keyword to search for on the server). Once the fetch(request) returns, the onComplete function is called and it iterates over each result, getting the content attribute (the only available attribute on an OpenSearchStore item) and puts into a table cell to be displayed on the HTML page.

**Note:** The version of the Dojo Toolkit within the Ajax client runtime package with Feature Pack for Web 2.0 currently does not handle finding the root of the search results by ID. For instance, setting the query parameter to something like “#searchResultTable” does not work.

## 9.12.4 IBM SOAP library

The IBM SOAP library is composed of principally of two parts:

- ▶ The SOAP service

This library extends the `dojo.rpc.RpcService` class and provides an easier way to create the SOAP envelope around a request.

- ▶ The SOAP service widget

This widget uses the SOAP service and provides a convenient way to connect to external SOAP services and invoke their methods.

### Package structure

The SOAP service and SOAP service widget are distributed in two different packages:

- ▶ The IBM WebSphere Application Server Feature Pack for Web 2.0 installer places the package structure outlined below into the folder located at `<app_server_root>/Web2fep/ajax-rt_1.X/ibm_soap`.
- ▶ The stand-alone package includes the given package structure inside the folder located at `/AjaxClientRuntime_1.X/ibm_soap`.

The IBM SOAP library package is organized in the structure shown in Figure 9-33.

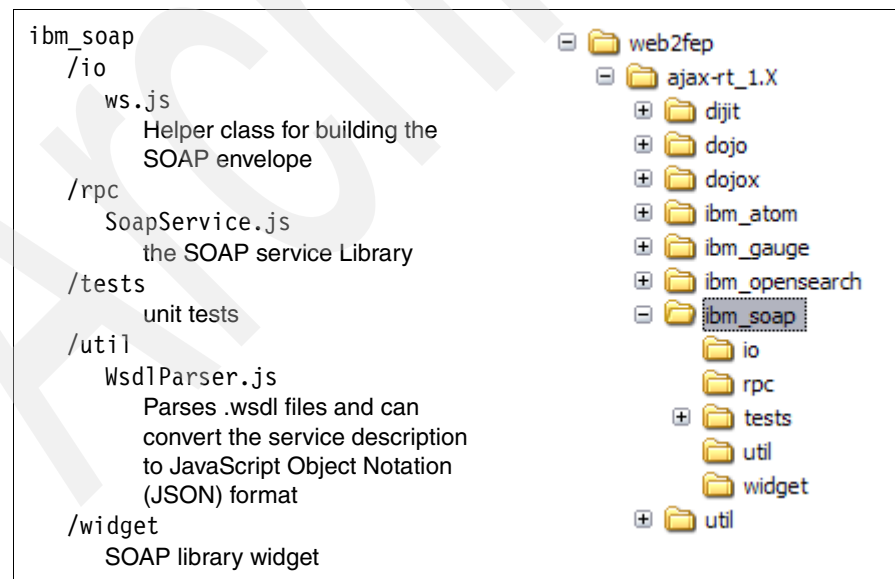


Figure 9-33 The IBM SOAP library package structure



## The SOAP service

The SOAP service can be instantiated using the service description from either a local or a remote file. The service description can be either in a .smd<sup>4</sup> file (`<div dojoType="ibm_soap.widget.SoapService" id="bnpriceService" url="/bnpriceGenerated.smd">`) or a .wsdl file format (`<div dojoType="ibm_soap.widget.SoapService" id="amazonCommerceService" url="/AWSCommerceService.wsdl">`).

You must provide the location of the service description as the URL. After the widgets are parsed and instantiated, the service is ready for use. Typically, you would begin with a service that provides a service description as a .wsdl file.

## WSDL parser

You can use the provided `WsdParser` to convert to an .smd format, as shown in Example 9-129.

*Example 9-129 Converting from WSDL to SMD*

---

```
var parser=new ibm_soap.util.WsdParser();
parser.parse("./serviceDescription.wsdl");
var smdString = parser.smdString;
```

---

After parsing a WSDL format description, you can access the results by using the `smdString` or the `smdObject` member of the parser. Consider the following items when converting the service description:

- ▶ Any complex type information contained in the service description is not converted to the JavaScript Object Notation (JSON) format. The type is considered as an object. Refer to the original Web Service Definition Language (WSDL) file to correctly form the parameters.
- ▶ The `WsdParser` accepts a WSDL description in one of the following ways:
  - URL

If the argument passed in to the `parse` method is a string, then that argument is assumed to be a URL to the .wsdl file:

```
parser.parse("./serviceDescription.wsdl")
```

---

<sup>4</sup> A Service Mapping Description (SMD) is a JSON representation describing Web services. The SMD specification is available at:  
<http://groups.google.com/group/json-schema/Web/service-mapping-description-proposal>

– Object with wsdlStr

If the argument is not a string, then the parser searches for a member named wsdlStr within the object passed in and, if found, parses it to retrieve the service description.

```
// Assume that the variable wsdlStr holds the
// WSDL description as a string
var args = new Object();
args.wsdlStr = wsdlStr
parser.parse(args);
```

To call a method described by the service, you need to know the method name and the structure of the parameters expected by the method. For instance, the following example details how to call the ItemLookup method of the Amazon service.

The required properties are set by calling the setPropertyValue method. The variable isbn stores the ISBN that is to be located. The service method is then invoked using the service member of the widget. That call returns an object of type dojo.Deferred that contains the BODY node of the returned content. You can use simple Document Object Model (DOM) manipulation to access the result.

*Example 9-130 Working with the IBM SOAP library*

---

```
// 1. Setting the required property values. Assume accessKeyId holds
// the AccessKeyId required by Amazon.
var serviceParms = new dojox.wire.ml.XmlElement("ItemLookupRequest");
serviceParms.setPropertyValue("AWSAccessKeyId",accessKeyId);
serviceParms.setPropertyValue("ItemId",isbn);

// 2. Invoke the service method
var deferred = amazonCommerceService.service.ItemLookup(serviceParms);

// 3. Access the callback object using DOM manipulation
deferred.addCallback(function(results) {
    var title =
        results.getElementsByTagName("Title")[0].firstChild.nodeValue;
    var author =
        results.getElementsByTagName("Author")[0].firstChild.nodeValue;
    var publisher =
        results.getElementsByTagName("Manufacturer")[0].firstChild.nodeValue;
    var resultNode = dojo.byId("details");
    resultNode.innerHTML = "<br><b>Title:</b>" + title
```

```
+ "<br><b>Author:</b>" + author  
+ "<br><b>Publisher:</b>" + publisher;  
});
```

---

### ***Proxy requirement***

Because of cross-domain XMLHttpRequest (XHR) restrictions, you might need to use a proxy (see 6.3, “Ajax Proxy” on page 131) to reach the service endpoints that are defined in the service description. You must modify the service endpoints to incorporate the proxy. The modified version of the Barnes & Noble price service might resemble the following example:

```
/proxy/http/www.abundanttech.com/WebServices/bnprice/bnprice.asmx
```

After you change the service endpoint URLs in the service description file, you might also need to perform additional configuration for the proxy to forward requests to the external server.

## **9.13 Adding the Web 2.0 FEP to existing applications**

This section presents an example of enhancing an existing J2EE application with the features provided in the IBM WebSphere Application Server Feature Pack for Web 2.0. We chose the sample application PlantsByWebSphere, which is among a number of samples provided with the IBM WebSphere Application Server Feature Pack for Web 2.0. The PlantsByWebSphere sample application represents a fictitious online plant store where one can order and purchase flowers, trees, vegetables, and accessories.

**Note:** Additional details on the Web 2.0 enhancements of the PlantsByWebSphere sample application are available in a paper by Kevin Haverlock at:

[http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/topic/com.ibm.iea.wasfpWeb20/wasfpWeb20/6.0/Overview/PBW\\_J2EE\\_Ajax\\_enhancements.pdf](http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/topic/com.ibm.iea.wasfpWeb20/wasfpWeb20/6.0/Overview/PBW_J2EE_Ajax_enhancements.pdf)

Figure 9-34 describes the architecture of the application in its original form before enhancing it with Ajax styled features. The architecture is intended to be fairly typical for a J2EE application running on WebSphere Application Server. At a high level, the application adheres to a Model-View-Controller design pattern. A browser accesses the URL for the application, which returns a JSP rendered HTML page. Additional requests are issued to the Web application from the browser and servlets are used to control the flow as users move through the purchase request. EJBs are used to serve model data available on the database.

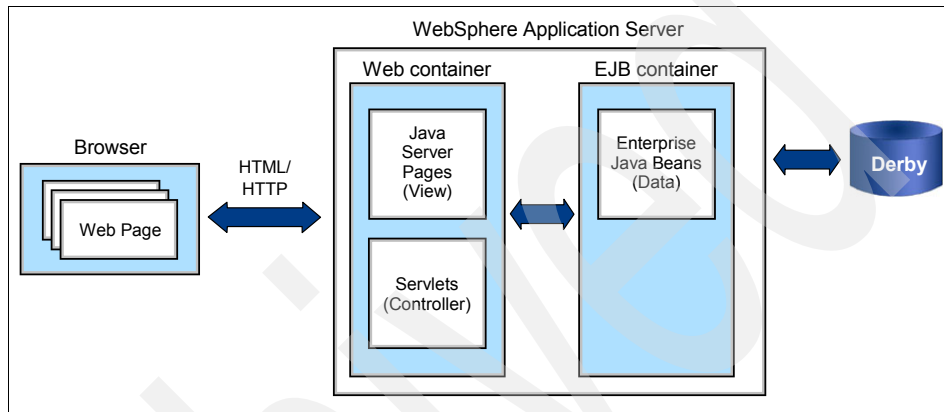


Figure 9-34 Enterprise Application Architecture: Traditional

Figure 9-35 shows how the architecture of the application has been augmented using Ajax. The intention is not to rewrite the application, but rather to take advantage of technologies in the IBM feature pack that would improve and create a more interactive and rich experience for the user.

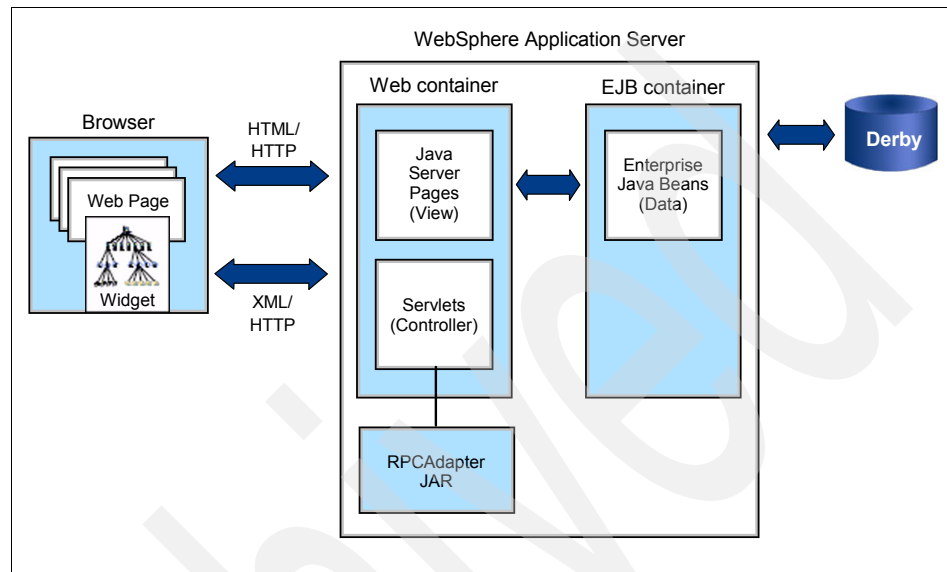


Figure 9-35 Enterprise Application Architecture: Web 2.0

In Figure 9-35, notice that enhancements have been made on the browser as well as the server side. On the browser side, the application now uses widgets provided with the Dojo Toolkit. Additionally, customized user interface widgets were created to improve the interactive nature of PlantsByWebSphere without having to rewrite it. The customized user interface widgets are asynchronous, meaning that they communicate using the browser's XHR mechanism supported by the Dojo Toolkit. The widgets use an XML interchange format to exchange data with the server. On the server, the RPCAdapter servlet provided with the feature pack is used to convert the EJB data into an XML interchange format that can easily be consumed by newly created widgets on the browser. A characteristic of Ajax applications is an improved response on the user interface. PlantsByWebSphere uses the Dojo Toolkit within the Web browser to improve the user interface of the application. The Dojo Toolkit is pure JavaScript, and the JavaScript files can be hosted directly in the Web-content directory of the Web Archive File (WAR) or exist as static Web content on a performance-optimized content delivery network. For this PlantsByWebSphere sample, the Dojo Toolkit JavaScript files are hosted as part of the WAR and the Dojo Toolkit widgets are directly embedded into the JSP pages.

### 9.13.1 Browser-side enhancement

In this section we discuss browser-side enhancement.

#### Web form handling

The Dojo Toolkit provides a rich set of form handling validation that can be added to Web pages. In this example we enhance the register.jsp page.

We can begin by declaring the usage of Dojo within our JSP page register.jsp. The first `<script>` tag in Figure 9-36 declares usage of `dojo.js`. The `dojo.js` is the core Dojo Toolkit kernel and is required when using the Dojo Toolkit.

```
<script type="text/javascript"
    src="/PlantsByWebSphereAjax/dojo/dojo.js"
    djConfig="isDebug: false, parseOnLoad: true,
    extraLocale: ['de-de', 'en-us']"></script>

<script type="text/javascript">
    dojo.require("dijit.form.TextBox");
    dojo.require("dijit.form.ValidationTextBox");
    dojo.require("dojo.parser");    // scan page for widgets
                                    //and instantiate them
</script>
```

Figure 9-36 Usage of Dojo Toolkit

The second `<script>` tag in Figure 9-36 declares the dojo widgets that the page uses. The `dojo.require` clause is comparable to the Java `import` or the C++ `includes` clause. The statement tells the `dojo.parser` where to locate the appropriate Dojo Toolkit JavaScript file that the page requires. For this example, the page uses the `dijit.form.ValidationTextBox` widget. The JavaScript `dojo.parser` needs to be included in the page as well. The parser is used to scan the page for widgets to declare.

Figure 9-36 shows how the Dojo Toolkit is declared within your JavaScript and the type of widgets the page requires. Figure 9-37 shows how a widget is declared within the page. The `dojo.parser` that was declared in Figure 9-36 will be used to scan the page and inline the necessary JavaScript code from the Dojo Toolkit to enable dynamic form validation.

```
<p><input type="text" name="fname" value="<%=fname%>"
    size="20" id="fname" dojoType="dijit.form.ValidationTextBox"
    lowercase="false" required="true"
    promptMessage="Enter First Name" /></p>
```

Figure 9-37 Dojo validation text box

Figure 9-38 shows the result. If the user fails to enter a value for a required field, the browser will prompt the user.

2. Contact Information

First Name \*

Last Name \*

Address Line 1 \*  ⚠ Enter Address

Address Line 2

City \*  ⚠

State \*  ⚠

Zip Code \*  ⚠

Phone (daytime) \*

Figure 9-38 Ajax validation form

## Adding custom widgets

Dojo provides a powerful framework to create innovative widgets that can be embedded directly within HTML pages. For our sample, to support the drag and drop function, a number of custom Dojo widget controls were created. These are `HtmlShoppingCart`, `InventoryGrid`, `InventoryItem`, and `ItemDetails`. The widgets include support for displaying the inventory in a grid, displaying detail information about the items, and dragging items to a shopping cart. When you click **check out**, the content of the shopping cart widget is sent to the server for processing. The grid widget that displays the content of the catalog derives the information by issuing a request to the server using the `dojo.xhr(Get,Put)`. The response from the server is sent back in XML format and contains detailed information and a universal resource locator (URL) reference to where the image is located.

Figure 9-39 shows the catalog browsing page for the PlantsByWebSphere application. The catalog page shows pictures of items across the top. When the user clicks one of the pictures, detailed information about the item is displayed at the bottom.

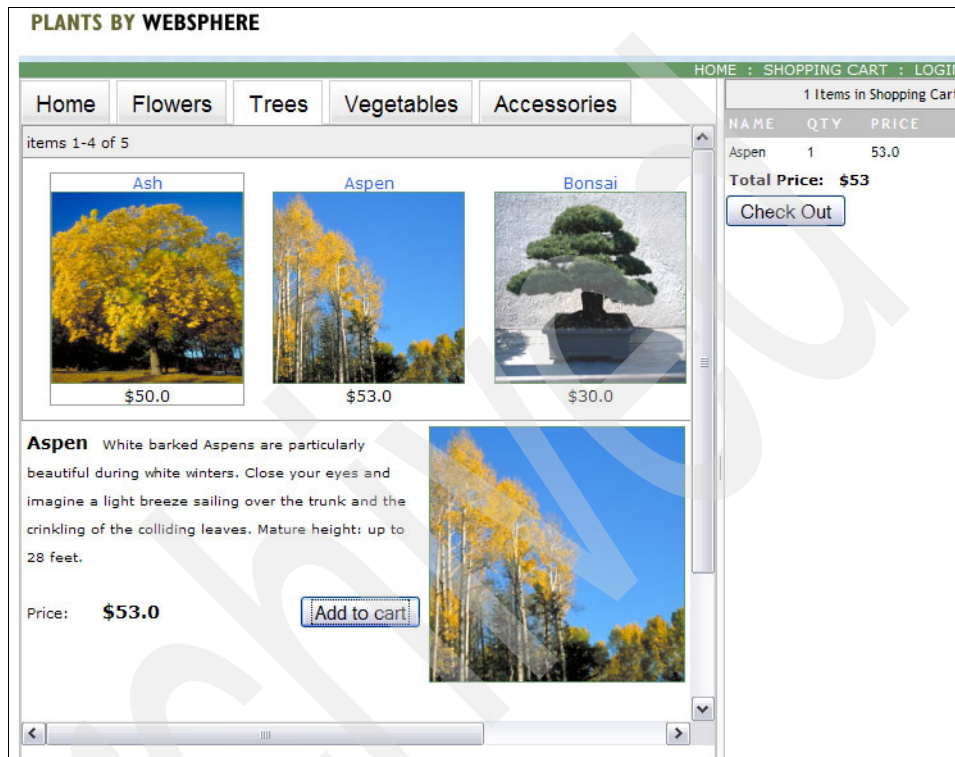


Figure 9-39 Shopping catalog



While this page consists of a number of different widgets, for now let us look at the ItemDetails Widget at the bottom in Figure 9-40. The itemDetail widget fetches information from the server and displays it in the details window. The user clicks the **Add to cart** button or drags the item to the cart using the left mouse button. The widget retrieves information from the server whenever the user clicks one of the catalog items at the top of the page.

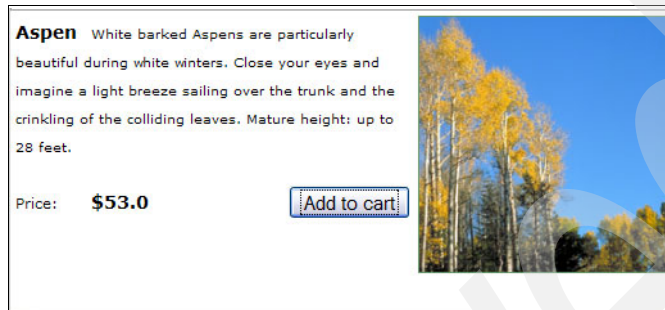


Figure 9-40 ItemDetails Widget

Let us look at how the ItemDetail widget was created. When creating widgets, there are three files one needs to consider. The first two are the HTML and CSS template files. These will be used to define the skeleton of how the page will look when the widget is rendered within the browser. In our sample these files can be found at /PlantsByWebSphere\_WEB/WebContent/ibm/widget/templates/. The HTML page will contain declared attachment points where the DOM elements will be inserted by your widget. The attachment points are called DojoAttachPoint.

Figure 9-41 shows a portion of the ItemDetails widget's HTML template and the DojoAttachPoint for the table rows. Since the JavaScript in the customized widget processes content, it will dynamically create elements and insert them into attachment points for the DOM.

```


|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |                                                                                                                                                                                                                                                                                        |                                                                                                                                                                                                                                                                                        |                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------|
| <span class="itemNameText" dojoattachpoint="nameElement">&lt;/span&gt; <span class="itemDescText" dojoattachpoint="descElement">&lt;/span&gt; <br&gt; <br&gt;="" <table="" cellpadding="0" cellspacing="0" height="100%" width="100%"> <tr style="width: 80%"> <td align="left"> <span style="font-family: verdana, arial, sans-serif; font-size: 11px;">Price:</span>&lt;/span&gt; </td> <td align="right"> <span &gt;\$&lt;span="" &gt;&lt;="" <="" class="itemNameText" dojoattachpoint="priceElement" span&gt;="" span&gt;&lt;="" td=""> <td align="right"> <span> button dojoType="dijit.form.Button" dojoAttachEvent="onclick: addToCart"&gt;Add to cart&lt;/button&gt; </span> </td> </span></td></tr> </br&gt;></span></span> | <span style="font-family: verdana, arial, sans-serif; font-size: 11px;">Price:</span> </span>                                                                                                                                                                                          | <span &gt;\$&lt;span="" &gt;&lt;="" <="" class="itemNameText" dojoattachpoint="priceElement" span&gt;="" span&gt;&lt;="" td=""> <td align="right"> <span> button dojoType="dijit.form.Button" dojoAttachEvent="onclick: addToCart"&gt;Add to cart&lt;/button&gt; </span> </td> </span> | <span> button dojoType="dijit.form.Button" dojoAttachEvent="onclick: addToCart"&gt;Add to cart&lt;/button&gt; </span> |
| <span style="font-family: verdana, arial, sans-serif; font-size: 11px;">Price:</span> </span>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         | <span &gt;\$&lt;span="" &gt;&lt;="" <="" class="itemNameText" dojoattachpoint="priceElement" span&gt;="" span&gt;&lt;="" td=""> <td align="right"> <span> button dojoType="dijit.form.Button" dojoAttachEvent="onclick: addToCart"&gt;Add to cart&lt;/button&gt; </span> </td> </span> | <span> button dojoType="dijit.form.Button" dojoAttachEvent="onclick: addToCart"&gt;Add to cart&lt;/button&gt; </span>                                                                                                                                                                  |                                                                                                                       |

 </div> |
```

Figure 9-41 HTML template of ItemDetails widget

The widget's template and CSS files together define the appearance of the widget and identify where in the DOM the updates will occur. The final step is to add the JavaScript code that provides the logic to the widget. Figure 9-42 shows the beginning section of the customized ItemDetail widget. The code begins by declaring the widget to be created:

```
dojo.provide("ibm.widget.ItemDetails")
```

```

dojo.declare(
  "ibm.widget.ItemDetails",
  [dijit.Widget, dijit._Templated],
  {
    initializer: function(){
    },
    templatePath: dojo.moduleUrl("ibm", "widget/templates/ItemDetails.html"),
    isContainer: false,
  }
);

```

Figure 9-42 Declaration of ItemDetails Widget

The name is used to reference the newly created widget. Below the `dojo.provide` is the `dojo.declare`. The `dojo.declare` defines the widget and any inheritance that this widget might have. In this case, the code declares inheritance from base `dijit._Widget` and `dijit._Templated`. Since this is a templated widget, the code declares the `templatePath`. The template path defines the location of the template that was defined above. The `dojo.moduleUrl` is a utility function that is used to resolve the location of the template.

Further down in the code, the `DojoAttachPoint` declarations defined in Figure 9-42 on page 430 are declared. These will be the root nodes of the DOM references that will be updated. In this case `nameElement` and `descElement` are declared, among a number of other variables. The data that the `ItemDetails` widget requires is located on the server. The Dojo Toolkit offers a powerful I/O transport mechanism that can be used to issue `XMLHttpRequests` (XHR) requests to the server and process the result. The XHR API opens an independent communication channel within the browser.

Figure 9-43 shows the use of the `dojo.xhrGet` function to retrieve the data from the server.

- ▶ The Dojo Toolkit wraps the XHR API with its own API, which simplifies working with the XHR API.
- ▶ The URL defines the address of the server.
- ▶ The `?p0="+item.id` is a URL parameter that is passed to the server.
- ▶ The timeout defines how long to wait before giving up on the connection to the server.
- ▶ The headers define additional HTTP headers that should be applied to the request sent to the server.
- ▶ The `handleAs` tells the Dojo Toolkit how to handle the response. In this case, the expected response is XML code.
- ▶ The `deferred.addCallback(function(response)` is the callback function that will be invoked by the Dojo Toolkit when the response is received from the server.
- ▶ The previous section covered the `DojoAttachPoint` as being the location of DOM node to be modified. The `descElement` is one of the DOM nodes modified here.
- ▶ The `self.descElement.innerHTML` value is set to the description text returned in the XML response. The `descElement` is the DOM node that was declared previously in the Template file of the widget.

```
var deferred = dojo.xhrGet( {  
    url: "/PlantsByWebSphereAjax/servlet/ShoppingServlet?action=addtocart&qty=1&itemID="+item.id,  
    timeout: 5000,  
    handleAs : "xml",  
    headers: { "Content-Type":"text/html" },  
    preventCache: true  
});  
deferred.addCallback(function(response){
```

*Figure 9-43 Use of `dojo.xhrGet`*

Once the customized Widget is created, it needs to be inserted into the HTML page. Figure 9-44 shows what the page looks like. This source is from the `trees.html` file. It shows a fragment of the HTML behind the page. The `DIV` tag is used to embed the widget and contains the keyword `dojoType` which declares the `ItemDetails` widget. There are additional parameters passed to the widget in the form of `url` which is the URL request to the `RPCAdapter` on the server. If you recall, the `ItemDetails` widget makes an XHR request to the server to retrieve the data to be displayed. The server side code will be covered in the next section which includes the usage of the `RPCAdapter`.

```
<div dojoType="ibm.widget.ItemDetails"
    url="/PlantsByWebSphereAjax/servlet/RPCAdapter/httprpc/Sample/detailRequest"
    style="width: 100%;" itemTopic="itemdetails_tree"></div>
</div>
```

Figure 9-44 Use of `ItemDetails` widget

Similarly, other custom widgets were created and included in the `index.html` page, as shown in Figure 9-45.

```
<script type="text/javascript">
    dojo.require("dijit.layout.ContentPane");
    dojo.require("dijit.layout.TabContainer");
    dojo.require("dijit.Tooltip");
    dojo.require("dijit.layout.LinkPane");
    dojo.require("dijit.form.Button");

    // scan page for widgets and instantiate them
    dojo.require("dojo.parser");

    dojo.require("ibm.widget.InventoryGrid");
    dojo.require("ibm.widget.InventoryItem");
    dojo.require("ibm.widget.ItemDetails");
    dojo.require("ibm.widget.HtmlShoppingCart");

    // dijit
    dojo.require("dijit.layout.LayoutContainer");
    dojo.require("dijit.layout.SplitContainer");

</script>
```

Figure 9-45 Importing other custom widgets

### 9.13.2 Server-side enhancement

The PlantsByWebSphere application also includes an additional adapter layer using the RPCAdapter. In our PlantsByWebSphere sample, the RPCAdapter is used to map client-side GET requests to EJB or servlet session data. The response is returned as XML data and used as input to the Dojo widgets within the browser. The data can easily be consumed and rendered by JavaScript-based clients using Dojo. The client-side widget issues an XHR request to the server for some data. The servlet interprets the request and matches it to the correct EJB.

The EJB is called and the EJB container issues a request to the database that returns the result. The result needs to be encoded into XML and returned back to the JavaScript widget on the browser. Figure 9-46 shows the entity and session beans available in PlantsByWebSphereAjax\_EJB module.

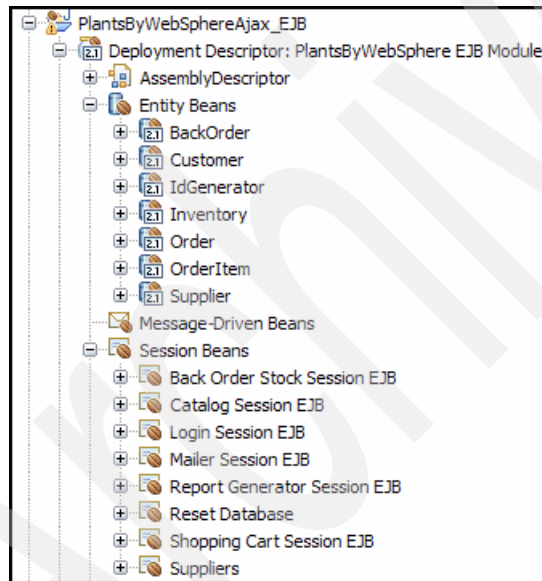


Figure 9-46 Enterprise Java Beans

To map these EJB methods to light-weight client-side GET requests, we need to have a POJO implemented. In our case the POJO implementation is contained in the `com.ibm.websphere.samples.plantsbyWebSpherewar.RpcConnector` class file. Figure 9-47 shows the POJO class that contains the implementations of the required methods.

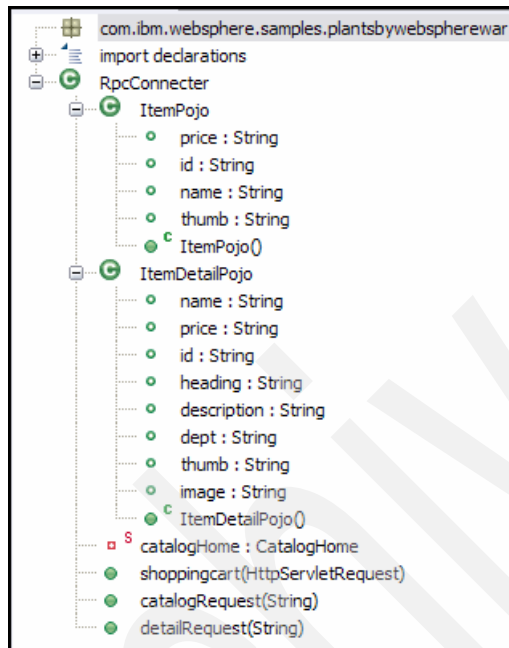


Figure 9-47 *RpcConnector* class structure

The RPCAdapter runtime JAR is included with the PlantsByWebSphere application and is configured using an XML file located in the WEB-INF of the WAR file. Figure 9-48 shows the RPCAdapterConfig.xml configuration file for RPCAdapter where all the methods of the POJO (RpcConnector) are mapped.

```
<?xml version="1.0" encoding="UTF-8"?>
<rpcAdapter>
  <default-format>xml</default-format>
  <services>
    <pojo>
      <name>Sample</name>
      <implementation>
        com.ibm.websphere.samples.plantsbywebspherewar.RpcConnector
      </implementation>
      <methods filter="whitelisting">
        <method>
          <name>shoppingcart</name>
          <description>returns back a value</description>
          <http-method>GET</http-method>
        </method>
        <method>
          <name>catalogRequest</name>
          <description>
            returns back contents of the PlantsByWebSphere
            catalog. Valid requests are 0-4
          </description>
          <http-method>GET</http-method>
          <parameters>
            <parameter>
              <name source="request">message</name>
              <description>
                Contains the message to be returned
              </description>
            </parameter>
          </parameters>
        </method>
      </methods>
    </pojo>
  </services>
</rpcAdapter>
```

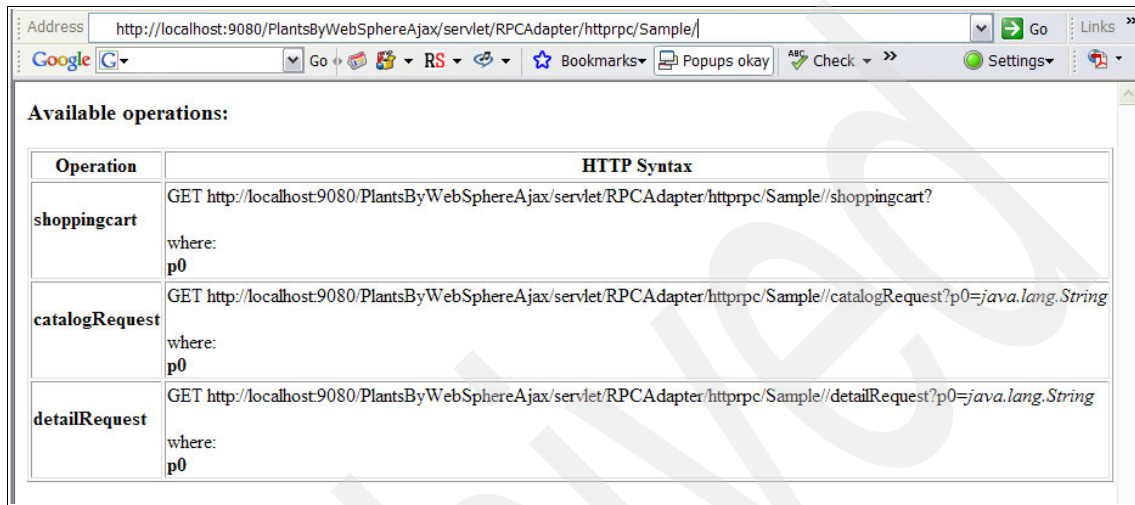
Figure 9-48 *RpcAdapterConfig.xml file*

Notice that the <default-format> returned by the RPCAdapter is declared as XML. The RPCAdapter also supports returning data in JSON form. The <implementation> element contains the user-defined class definition that the RPCAdapter will instantiate to invoke the method. The method name to invoke within the class is defined by the <name> element. The <http-method> that the RPCAdapter is expecting is the GET. The parameter value expected on the URL request is 'message'.



Once the required methods are configured a call to the server specifying the target as the following, the results displays is a list of all the methods configured for this sample POJO:

<http://localhost:9080/PlantsByWebSphereAjax/servlet/RPCAdapter/httprpc/Sample/>



Operation	HTTP Syntax
shoppingcart	GET http://localhost:9080/PlantsByWebSphereAjax/servlet/RPCAdapter/httprpc/Sample//shoppingcart? where: p0
catalogRequest	GET http://localhost:9080/PlantsByWebSphereAjax/servlet/RPCAdapter/httprpc/Sample//catalogRequest?p0= <i>java.lang.String</i> where: p0
detailRequest	GET http://localhost:9080/PlantsByWebSphereAjax/servlet/RPCAdapter/httprpc/Sample//detailRequest?p0= <i>java.lang.String</i> where: p0

Figure 9-49 *HttpRpc URLs mapped to EJB methods*

The method `detailrequest()` returns back a collection of POJO elements. The collection class is part of the `java.util` package. The POJO elements are derived from data returned by invoking the `StoreItem` EJB. The `StoreItem` is retrieved from the `catalog` EJB. The `catalog` EJB is a collection of the items that are contained within the `PlantsByWebSphere` inventory located in the `derby` database. Once a Java collection is returned, the `RPCAdapter` will transparently map the collection to XML data. The collection's key corresponds to the XML element and the key's value to the XML value. The XML stream is returned to the `ItemDetail` widget. By using the `RPCAdapter` and having it return XML, you have in effect created a service that others can connect to as well. The service only returns data, and it is up to the caller to figure out how to render the data. A usage scenario might be a partner company of `PlantsByWebSphere` that aggregates data from its own greenhouse with the catalog information maintained by `PlantsByWebSphere`. This scenario is sometimes referred to as a *mashup*.

## 9.14 Sample: PlantsByWebSphere application

The PlantsByWebSphere application is intended to be representative of a Web application that contains model, presentation, and control layers. The PlantsByWebSphere application is a fictional Web site that makes available nursery items such as vegetables, flowers, accessories, and trees. You can view an online catalog, select items, and add them to a cart. When the cart contains items, you can proceed to log in, supply credit card information, and check out.

This application serves as an example of a Web application that someone might attempt to develop. This application contains model, presentation, and controller layers that can be enhanced. Additionally, Ajax-style architecture is added through the JavaScript Dojo Toolkit. Specifically, the use of various Dojo widgets and enablement of drag-and-drop capabilities for the shopping cart are added.

### 9.14.1 Installation and configuration

The steps needed to install and then configure the PlantsByWebSphere sample application provided with the WebSphere Application Server Feature Pack for Web 2.0 utilizing the Rational Application Developer are:

1. Open Rational Application Developer in a new workspace and close the welcome page.
2. Click **File** → **New** → **Other** from the workbench toolbar.
3. Expand **Examples** → **IBM WebSphere Application Server Feature Pack for Web 2.0 samples** in the pop-up window.

4. Select **PlantsByWebSphere Sample** and click **Next**, as shown in Figure 9-50.

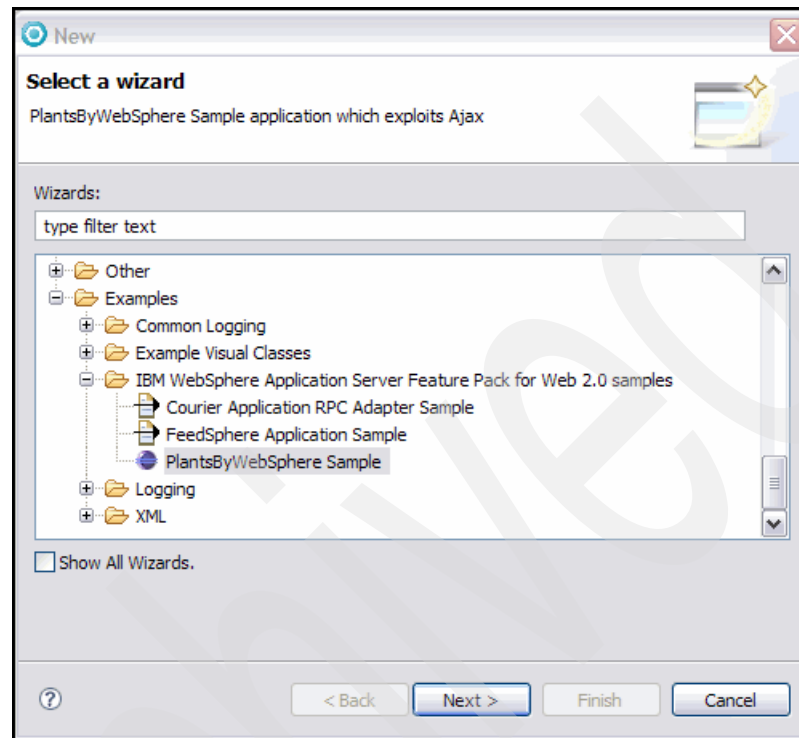


Figure 9-50 New project wizard

5. Select **WebSphere Application Server v6.1** as the target run time from the drop-down list. We will deploy this sample to the WebSphere test environment that was previously installed along with the installation of Rational Application Developer. Click **Next**, as shown in Figure 9-51.

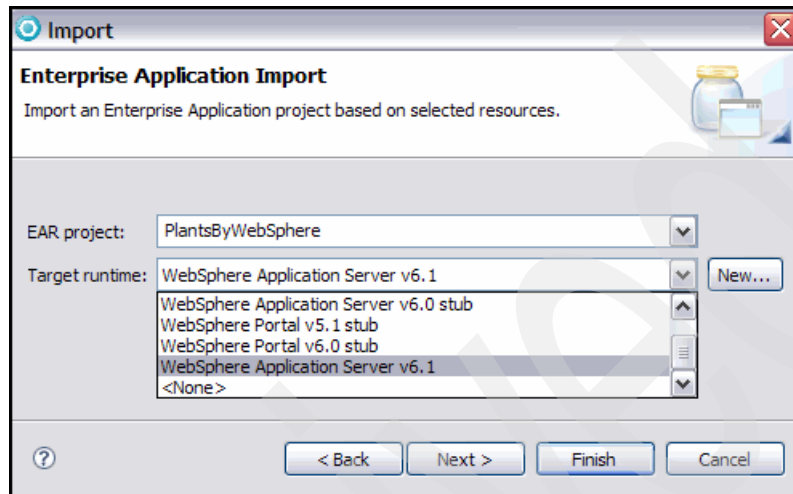


Figure 9-51 Import Enterprise Application

6. You can either import the utility jars required for this sample as utility projects or import them as utility JARs within the application EAR file. Click **Next**.

7. If needed, you can change the names of projects where EAR Modules will be imported. Click **Finish**, as shown in Figure 9-52.

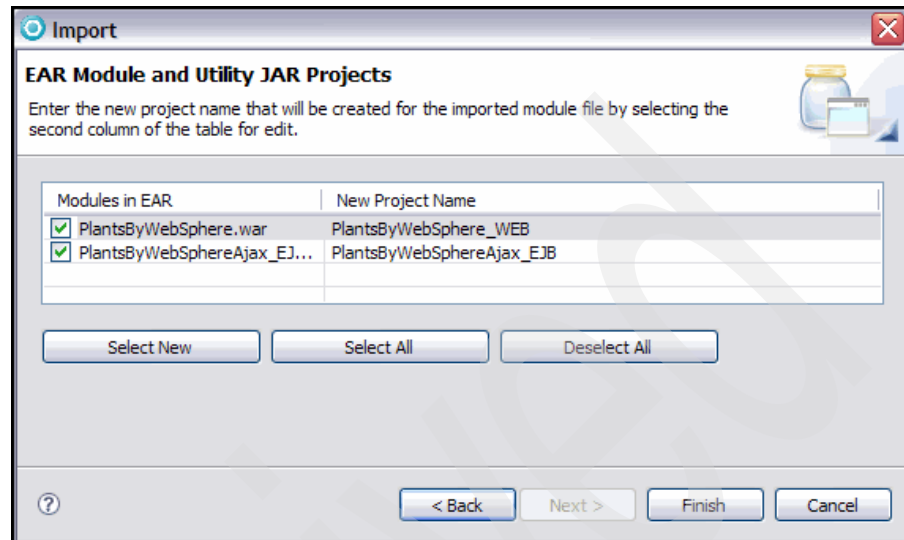


Figure 9-52 Supported modules for EAR

8. The complete EAR project with all its required modules will be imported. You can browse through the modules (PlantsByWebSphere - EAR module, PlantsByWebSphere\_Web - WEB module, PlantsByWebSphereAjax\_EJB - EJB module) in a Project Explorer view, as shown in Figure 9-53.

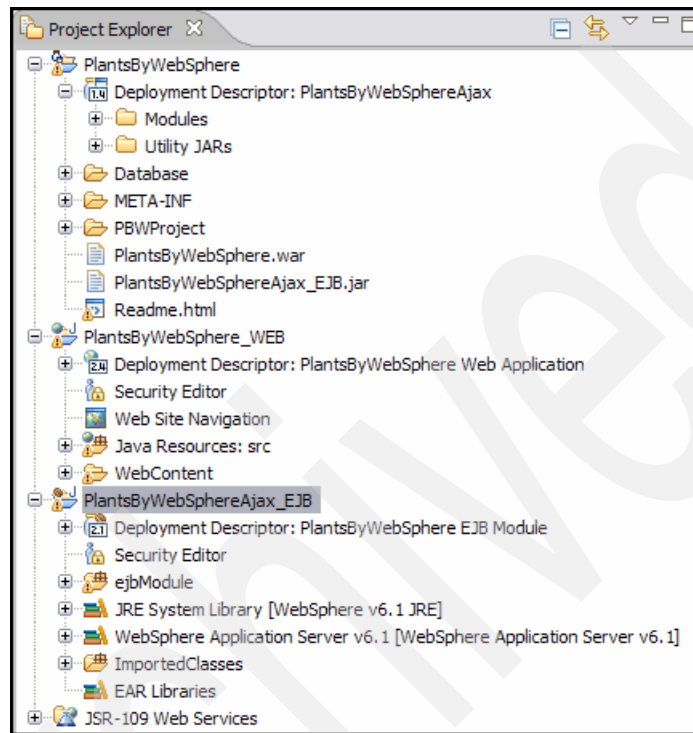
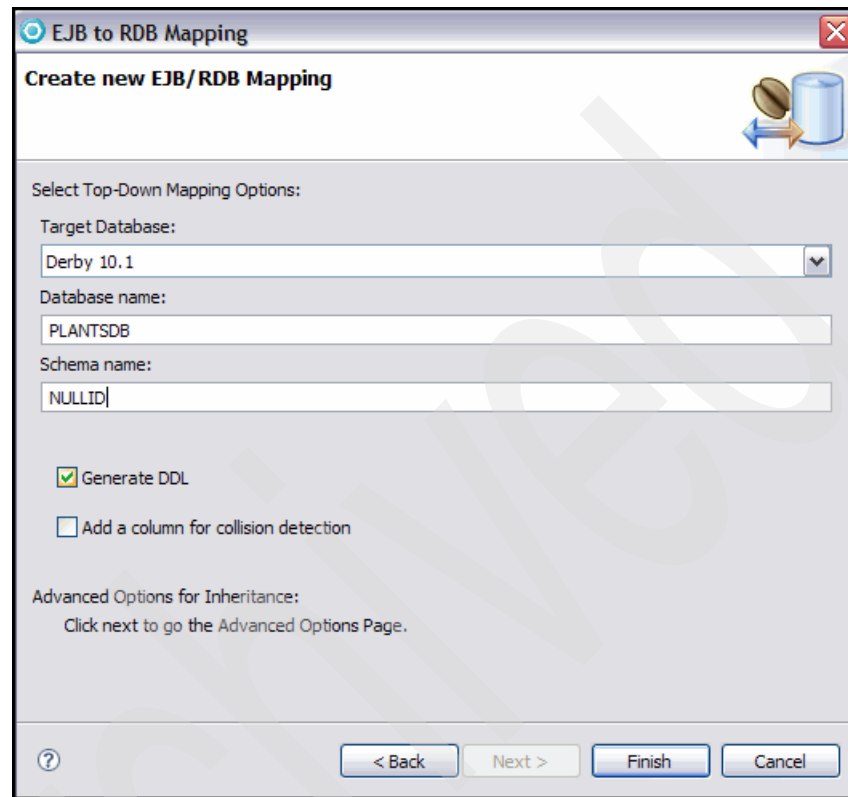


Figure 9-53 Project structure

9. Expand the **PlantsByWebSphereAjax\_EJB** module. Right-click **Deployment Descriptor** and select **EJB to RDB Mapping** → **Generate Map**.
10. You will see a EJB to RDB Mapping pop-up page with the option **Create a new backend folder** already selected. Click **Next**.
11. Select the **Top-Down** approach of mapping and click **Next**.

12. Select **Derby 10.1** as the target database, enter the database name as **PLANTSDB**, and click **Finish**, as shown in Figure 9-54.



The image shows a dialog box titled "EJB to RDB Mapping" with a subtitle "Create new EJB/RDB Mapping". It features a "Select Top-Down Mapping Options:" section with three text fields: "Target Database:" (set to "Derby 10.1"), "Database name:" (set to "PLANTSDB"), and "Schema name:" (set to "NULLID"). Below these are two checkboxes: "Generate DDL" (checked) and "Add a column for collision detection" (unchecked). An "Advanced Options for Inheritance:" section contains the text "Click next to go the Advanced Options Page." At the bottom are buttons for "< Back", "Next >", "Finish", and "Cancel".

Figure 9-54 EJB to Relational Database Mapping

You should now see a map file generated for the EJB and the database tables, as shown in Figure 9-55.

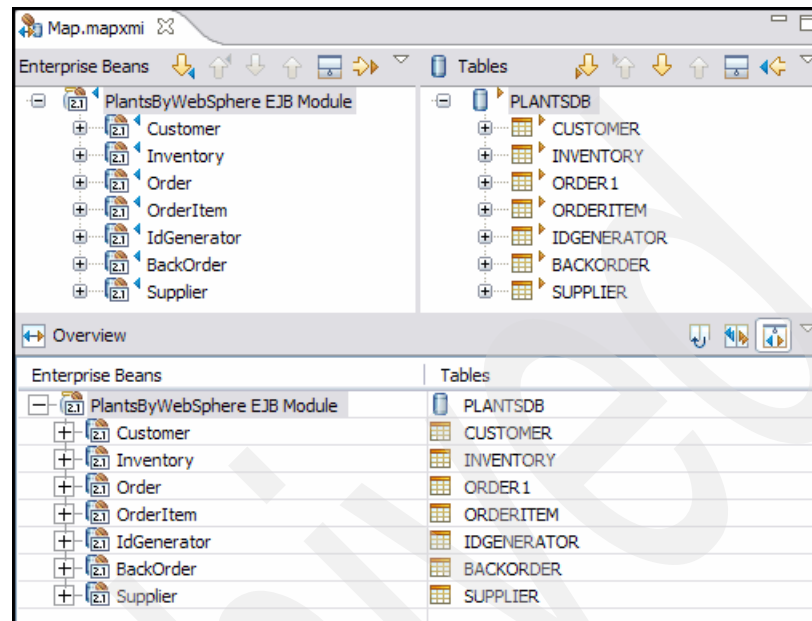


Figure 9-55 EJB to Relational Database Mapping Map

13. Open to the Servers view. Right-click **WebSphere Application Server v6.1** in the Servers view and click **Start**. When the server is successfully started the server status will change to started.
14. To configure the Derby database in the WebSphere Application Server, right-click **Application Server** in the Servers view and select **Run administrative console**.
15. Select **Resources** → **JDBC™** → **JDBC Providers** in the left pane. The JDBC provider encapsulates the implementation class for the specific database provider.
16. Select the scope. The scope is **Node=<Node Name>,server=<Server Name>**.
17. Select **New**.
18. For the database type select **Derby**.
19. For the provider type select **Derby JDBC Provider**.
20. For the implementation type select **XA Data source**.



21. For the name type select **Derby Provider for PlantsByWebSphereAjax (XA)**. Click **Next**, as shown in Figure 9-56.

Create a new JDBC Provider

Create a new JDBC Provider

→ Step 1: Create new JDBC provider

Step 2: Enter database class path information

Step 3: Summary

Create new JDBC provider

Set the basic configuration values of a JDBC provider, which encapsulates the specific vendor JDBC driver implementation classes that are required to access the database. The wizard fills in the name and the description fields, but you can type different values.

Scope  
cells:GOYALNode01Cell:nodes:GOYALNode01:servers:server1

\* Database type  
Derby

\* Provider type  
Derby JDBC Provider

\* Implementation type  
XA data source

\* Name  
Derby Provider for PlantsByWebSphereAjax(XA)

Description  
Derby embedded XA JDBC Provider. This provider is only configurable in version 6.0.2 and later nodes

Next Cancel

Figure 9-56 New JDBC Provider in administration console

22. Click **Finish**.
23. Select **Save** to save directly to the master configuration.
24. Create and associate the data source for the PlantsByWebSphere application.
25. Click **Resources** → **JDBC** → **Data sources** in the left menu.
26. Select the scope of **Node=<Node Name>,server=<Server Name>**.
27. Click **New** to create a new data source.
28. Enter PLANTSDB as the data source name.

29. For the JNDI name enter `jdbc/PlantsByWebSphereAjaxDataSource`. Click **Next**, as shown in Figure 9-57.

**Enter basic data source information**

Set the basic configuration values of a data source for association with your JDBC provider. A data source supplies the physical connections between the application server and the database.

Requirement: Use the Data sources (WebSphere(R) Application Server V4) console pages if your applications are based on the Enterprise JavaBeans (TM) (EJB) 1.0 specification or the Java(TM) Servlet 2.2 specification.

Scope  
cells:GOYALNode01Cell:nodes:GOYALNode01:servers:server1

\* Data source name  
PLANTSDB

\* JNDI name  
jdbc/PlantsByWebSphereAjaxDataSource

Component-managed authentication alias and XA recovery authentication alias  
Select a component-managed authentication alias. The selected authentication alias will also be set as the XA recovery authentication alias if your JDBC Provider supports XA. If you choose to [create a new J2C authentication alias](#), the wizard will be canceled.

(none) ▼

Figure 9-57 Configure data source

30. Select the radio button **Select an existing JDBC Provider**.
31. Select **Derby Provider for PlantsByWebSphereAjax (XA)**, which you defined earlier, and click **Next**.
32. For the database name enter `${APP_INSTALL_ROOT}/${CELL}/PlantsByWebSphereAjax.ear/Database/PLANTSDB`, which in our example is `C:\workspace\PlantsByWebSphere\Database\PLANTSDB`. Click **Next**.
33. Click **Finish**.
34. Click **Save** to save to the master configuration.
35. Right-click **WebSphere Application Server v6.1** in the Servers view and select **Add and Remove Projects**.

36. Select **PlantsByWebSphere** in the list of Available projects and click **Add**. The project will be moved into Configured projects. Click **Finish**, as shown in Figure 9-58.

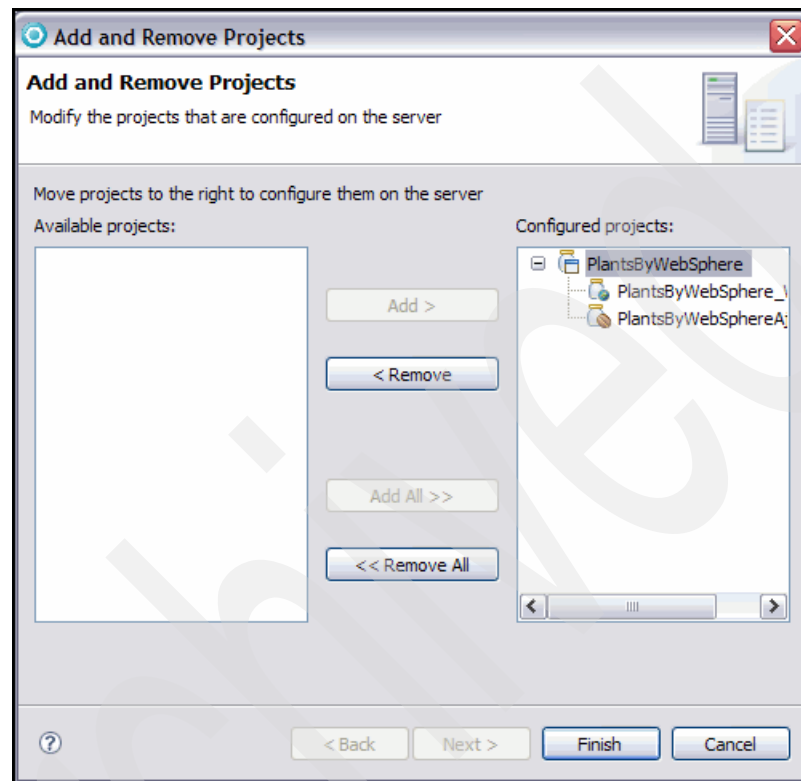


Figure 9-58 Add and remove projects to publish on test environment

The PlantsByWebSphere sample application is now published to the application server.

37. Right-click the **PlantsByWebSphere\_WEB** project in Project Explorer and select **Run As** → **Run on Server**.
38. Select **Finish** in the Run On Server dialog. A browser will now open with the home page of the sample application:
- <http://localhost:9080/PlantsByWebSphereAjax/>

The Plants By WebSphere sample application is now up and running successfully.

## 9.14.2 Usage

In this section we discuss usage.

### Catalog and shopping cart

The catalog and shopping cart are the starting points of the application when first connecting to the PlantsByWebSphere site. The shopping cart displays a set of tabs across the top that display various categories of items. The center page displays the contents of the category. You can scroll left or right by selecting **Back** or **Next**. When selecting an item from the catalog, the details page at the bottom is updated to provide additional information about the item.

To add the item to the shopping cart, you can drag the item from the catalog or click **Add to cart** in the item details section, as shown in Figure 9-59.

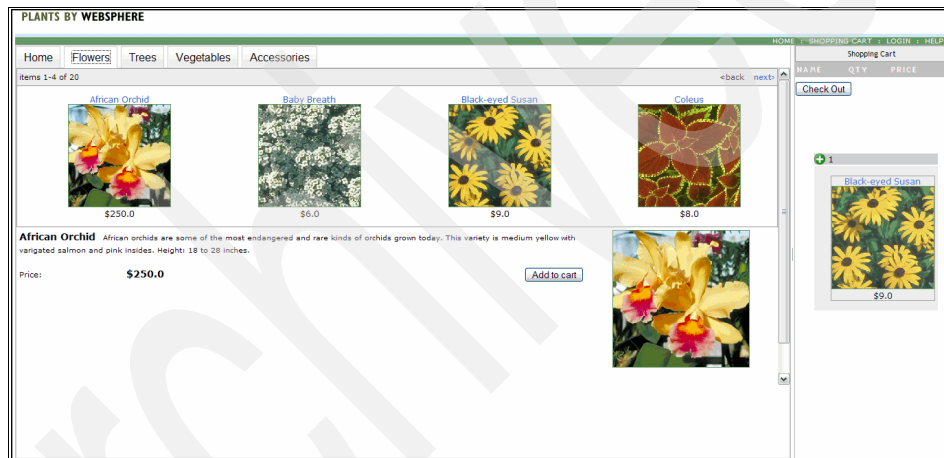


Figure 9-59 Shopping catalog

## Order processing

After items are added to the shopping cart from the catalog you can check out by clicking the **Checkout** button. The Ajax control issues a POST, which sends the product IDs and quantity for each item back to the server. The application then displays the checkout window, as shown in Figure 9-60.

### Shopping Cart

Here are the items you have selected. To recalculate your total after changing the quantity of an item, select the 'Recalculate' button. To remove an item from your cart, enter "0" as the quantity. Select 'Checkout Now' to begin the checkout process.

ITEM #	ITEM DESCRIPTION	PACKAGING	QUANTITY	PRICE	SUBTOTAL
F0003	<b>Black-eyed Susan</b>	2 plants	<input type="text" value="1"/>	\$9.00	\$9.00
A0001	<b>Bulb Digger</b>	Assembled	<input type="text" value="1"/>	\$12.00	\$12.00
T0003	<b>Bonsai</b>	0.5 gallon mature tree	<input type="text" value="1"/>	\$30.00	\$30.00

**Order Subtotal:\$51.00**

Figure 9-60 Shopping cart

After making adjustments to any quantities of items and confirming the order, you can proceed to order processing. If you are not logged in, you are directed to do so. The order processing page collects the user's name and shipping address. The order processing uses a Dojo control for the form validation. After placing the order, you will receive a confirmation page, as shown in Figure 9-61.

### Checkout

Enter the billing and shipping information for your order below. After you have completed all the required fields, click the "Checkout Now" button.

#### 1. Billing Address

Full Name \*

Address Line 1 \*

Address Line 2

City \*

State \*

Zip Code \*

Phone (daytime) \*

#### 2. Shipping Information

☒ Check here if the shipping address is the same as the billing address.

Full Name \*

Address Line 1 \*

Address Line 2

City \*

State \*

Zip Code \*

Phone (daytime) \*

#### 3. Shipping Method

Select a shipping method below.

Shipping Method \*

Credit Card \*

Card Number \*

Expiration Month \*

Expiration Year \*

Cardholder Name \*

#### 4. Your Order

Below is your final order. To make any changes to your order, click the "Continue Shopping" button below.

ITEM #	ITEM DESCRIPTION	PACKAGING	QUANTITY	PRICE	SUBTOTAL
F0003	Black-eyed Susan	2 plants	1	\$9.00	\$9.00
A0001	Bulb Digger	Assembled	1	\$12.00	\$12.00
T0003	Bonsai	0.5 gallon mature tree	1	\$30.00	\$30.00
Order Subtotal:					\$51.00
Shipping, Next Day Air:					\$12.99
<b>Order Total:</b>					<b>\$63.99</b>

Figure 9-61 Order confirmation

Figure 9-62 shows the order review page.

### Review Your Order

Review your order below and select 'Submit Order' at the bottom to place your order. You can also add more items to your order by selecting 'Continue Shopping'.

#### Order Information

ORDER TOTAL	SHIPPING ADDRESS	BILLING ADDRESS
<b>\$63.99</b>	Ankur Goyal IBM India Pvt. Ltd. , Birla Towers Connaught Place New Delhi, DELHI 11001 9818519931	Ankur Goyal IBM India Pvt. Ltd. , Birla Towers Connaught Place New Delhi, DELHI 11001 9818519931

#### Order Details

ITEM #	ITEM DESCRIPTION	PACKAGING	QUANTITY	PRICE	SUBTOTAL
F0003	Black-eyed Susan	2 plants	1	\$9.00	\$9.00
A0001	Bulb Digger	Assembled	1	\$12.00	\$12.00
T0003	Bonsai	0.5 gallon mature tree	1	\$30.00	\$30.00

Order Subtotal:

\$51.00

Shipping, Next Day Air:

\$12.99

Order Total:

**\$63.99**

Continue Shopping

Submit Order

Figure 9-62 Review order

## User registration

If this is the first time that you are accessing the page, you will need to register, as shown in Figure 9-64 on page 453.

### Registration

Enter the information below to set up your account. This information will not be shared without your permission. With your permission we will only share your name and email address with our trusted business partners.

Required fields are denoted with a red asterisk (\*).

#### 1. Login Information

E-mail address \*

Password \*

Verify Password \*

#### 2. Contact Information

First Name \*

Last Name \*

Address Line 1 \*

Address Line 2

City \*

State \*

Zip Code \*

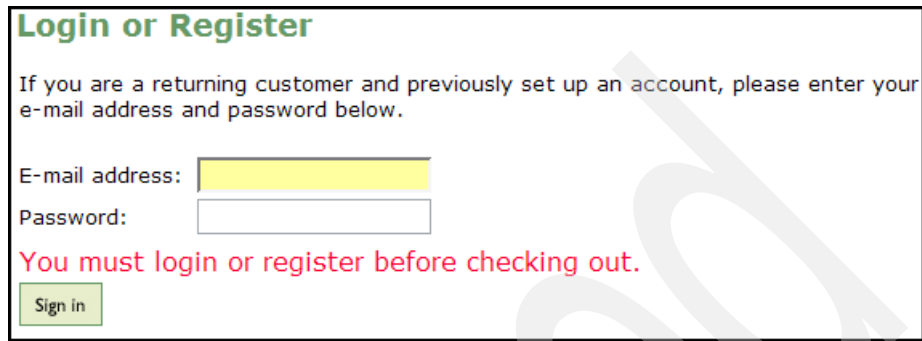
Phone (daytime) \*

Figure 9-63 Registration page



## User login

If you are not logged in, you are directed to a login page, as shown in Figure 9-64.



**Login or Register**

If you are a returning customer and previously set up an account, please enter your e-mail address and password below.

E-mail address:

Password:

You must login or register before checking out.

Figure 9-64 Login or Register page



## Web messaging service

In this chapter we introduce the Web messaging service, which can be used for connecting Asynchronous JavaScript and XML (Ajax) clients to near-real-time updated data such as stock quotes, an instant messaging application, or any scenario where dynamic data needs to be sent from server to browser.

The Quote Streamer sample application, which uses the Web messaging service to simulate stock quotes to a Dojo-enabled client application, is also described.

## 10.1 Introduction

The Web messaging service is a publish and subscribe implementation that connects the browser to the WebSphere Application Server service integration bus (SIBus) for server-side event push. The client/server communication is achieved through the Bayeux protocol, with the client-side support provided by the Dojo Toolkit.

The following sections outline the evolution of event-driven Web applications, introduce the Web messaging service, describe using the Web messaging service, and demonstrate developing Web messaging enabled applications.

## 10.2 Evolution of event-driven Web applications

In recent years, the Web platform has evolved along with the increased use of Internet technologies. This evolution involved a dramatic shift in the usage and understanding of these technologies, leading to the coining of the terms Web 1.0 and Web 2.0.

Web 1.0 was defined as being about connecting systems and making technology more efficient for computers. This focus has now shifted to what is now described as Web 2.0—connecting people and making technology efficient for computer users. This radical transfer of attention changes the way that information is delivered and presented, and how Web applications can be built.

### 10.2.1 Web applications circa 1997

One of the most identifiable attributes of Web 1.0 applications was that Web pages displayed in mostly static content. Delivery of new events to the browser required pressing the browser refresh button or submitting an HTML form and viewing the result.

A META HTML tag was available to allow Web pages to automatically refresh for scenarios where the application expected that updated information would be available:

```
<meta http-equiv="refresh" content="10" />
```

This refreshed the page every 10 seconds. There are significant issues with this approach. The entire page must always be reloaded even if the underlying data has not changed, meaning redundant and spurious refreshing of static content. This approach is also not performant, as the service is locked into handling full

page refreshes for each user at a fixed reload rate. Reloading the entire page also disrupts the user interface, resulting in the user not being able to interact with the Web application while the browser reloads the page. This classic Web application model is shown in Figure 10-1.

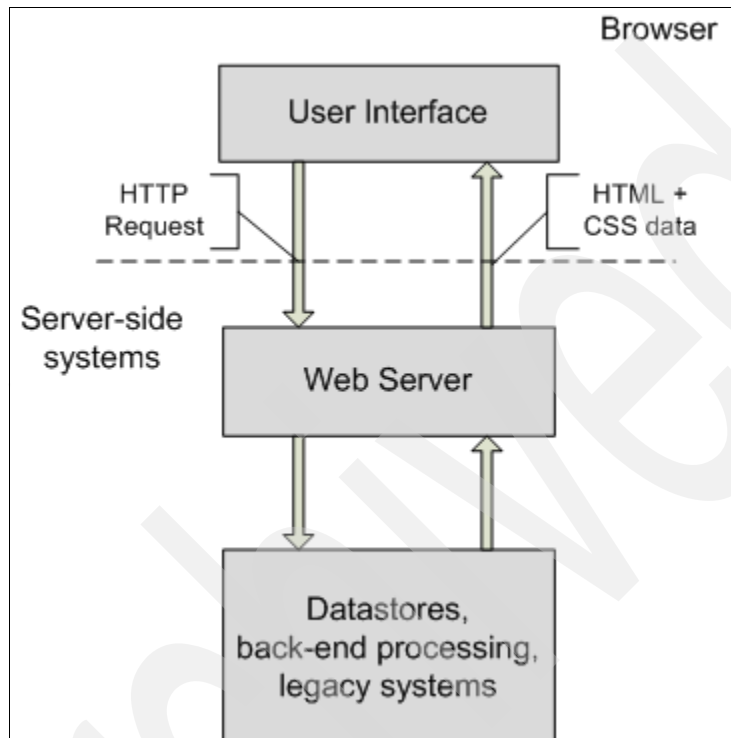


Figure 10-1 Classic Web application model

## 10.2.2 Web applications circa 2005

By 2005, Ajax technology was developed to support a different style of Web application. This approach permitted the Web application to work asynchronously, to be data-driven rather than presentation-driven, and to utilize XML over HTTP. The Ajax Web application model is shown in Figure 10-2.

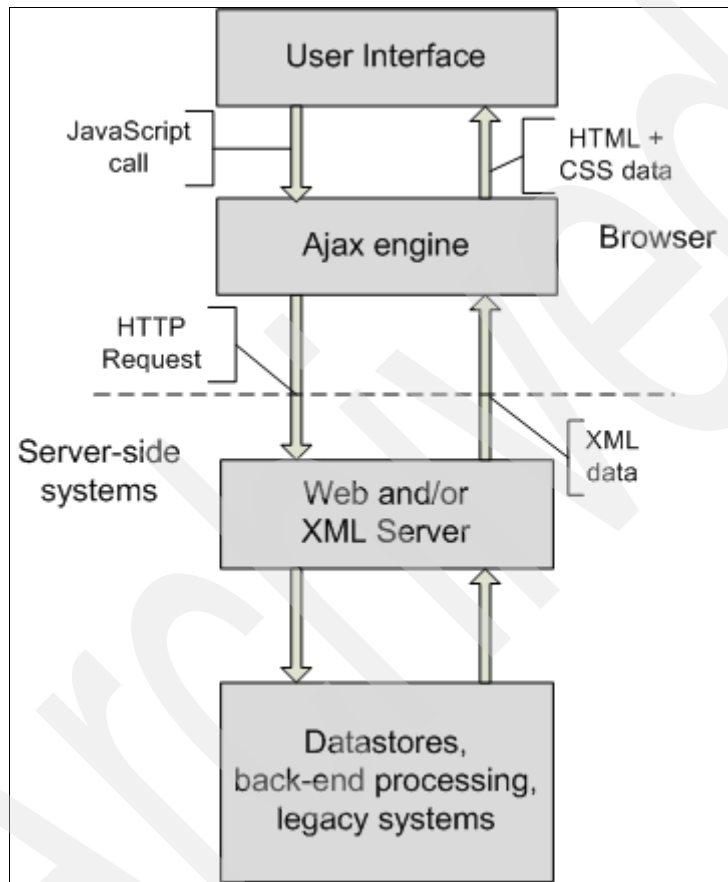


Figure 10-2 Ajax Web application model

An intermediary Ajax engine can be used to for user actions that do not require server-side functionality. If the browser-based application needs to contact the server, the Ajax engine does this asynchronously without hindering the user's interaction with the Web application. The diagram of this asynchronous interaction pattern is shown in Figure 10-3.

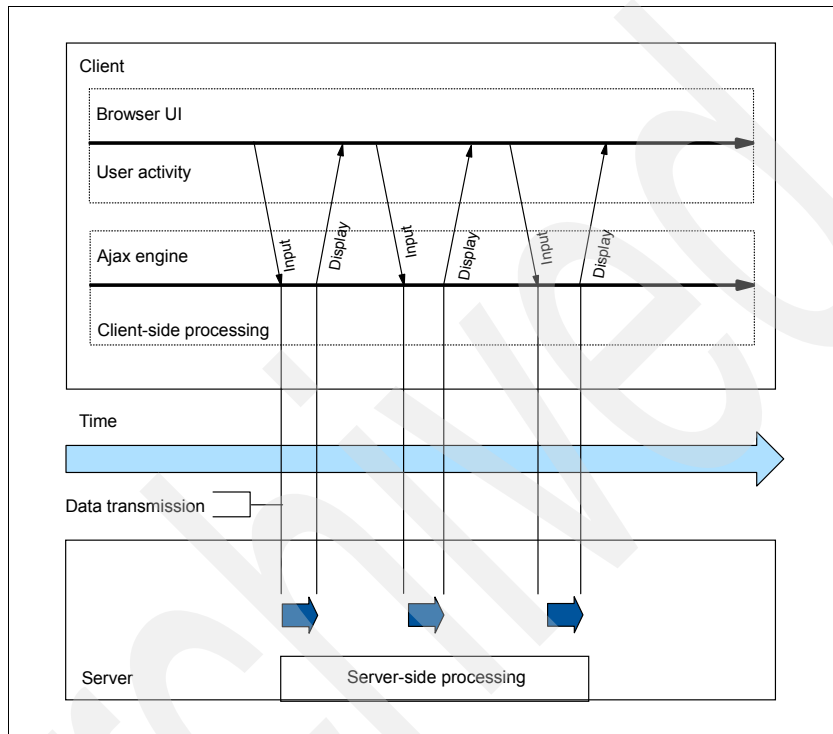


Figure 10-3 Asynchronous interaction pattern of Ajax application

However, problems exist with this method of pushing events from the server to the browser, potentially resulting in stale data being displayed for long-lived pages. As a workaround, Ajax polling is used. See Figure 10-4.

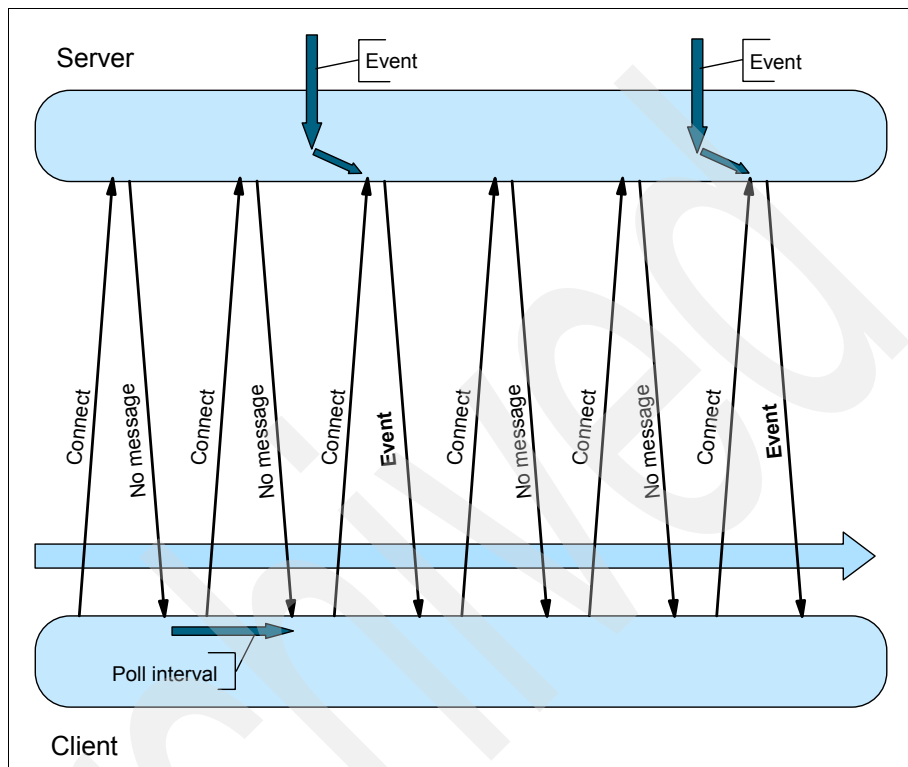


Figure 10-4 Ajax polling

### Issues with Ajax polling

There are the following issues with Ajax polling:

- ▶ Overhead is incurred when transporting and processing the *no message* HTTP request/response message:
  - Server side
  - Client side
  - Network



- ▶ Latency of event delivery
  - A fundamental design decision needs to be made on the polling interval. If too frequent, the resources are overworked. If too infrequent, the delivered information triggered by the event may be already stale.
  - From the time of event delivery, a client/server round-trip must be made for the event to reach the client.
- ▶ Scalability

A large number of clients or a low latency design (meaning frequent polling) will not scale well.

## 10.3 Comet

Comet was coined by Alex Russell in a March 2006 blog post<sup>1</sup> entitled “Comet: Low Latency Data for the Browser” to describe event-driven, server-push-based data streaming to browsers.

---

<sup>1</sup> <http://alex.dojotoolkit.org/?p=545>

Comet allows low-latency delivery of events to the browser in a much more timely manner than previous Ajax polling approaches. See Figure 10-5.

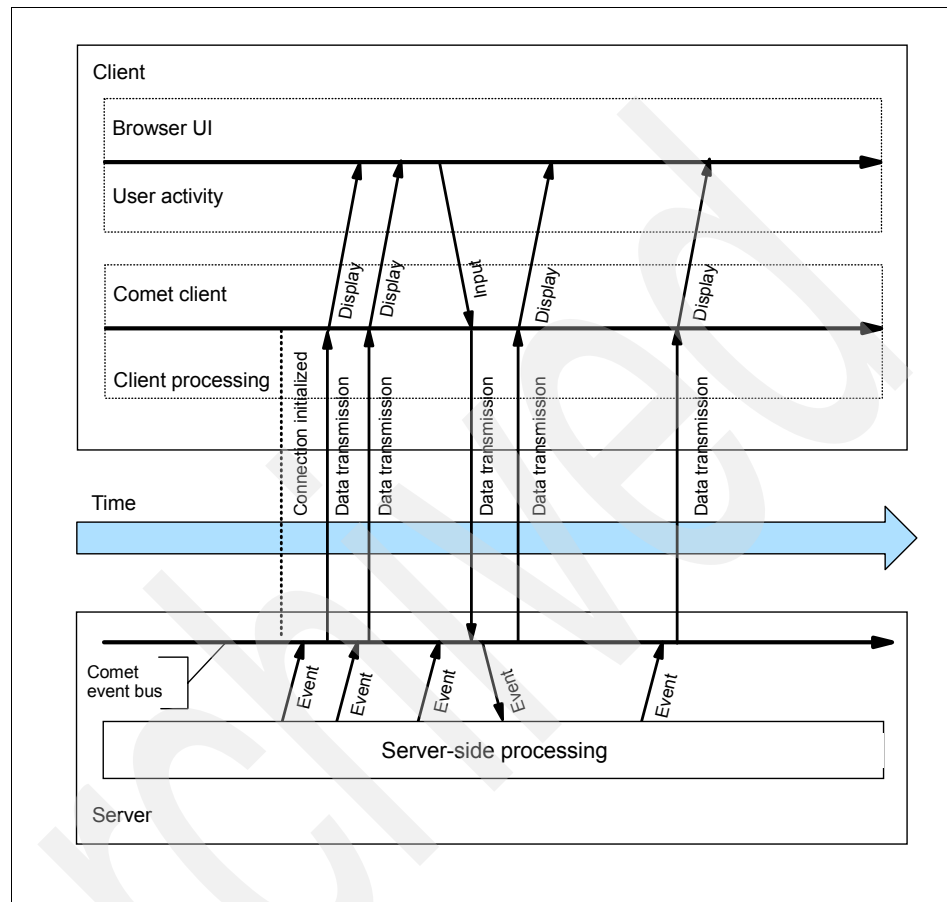


Figure 10-5 Comet Web application model

This long-polling approach does not require browser plug-ins or fat clients. It is capable of being scaled to large numbers of users using event-driven I/O.

Comet is an event-driven architecture on both the client and the server, providing a natural fit with a service-oriented architecture (SOA). In essence, Comet is any technique that allows a Web server to push near real-time events to a browser client. This technique has existed for some time under a variety of names (pushlets, server push, HTTP push, and Ajax push).

As Ajax improves single-user responsiveness, Comet improves collaborative, multi-user, event-driven application responsiveness. Information is delivered to the user at any time, not just in response to the user's input.

## 10.4 Bayeux Protocol

Bayeux is an HTTP-based messaging routing protocol for Comet. The protocol uses JSON for clients to subscribe to events and for servers to deliver them in a more timely manner, which is possible with Ajax-based polling. The protocol is designed such that any Bayeux client can communicate with any Bayeux server, on either a permanent connection created using a handshake or by sending simple single messages over channels.

All Bayeux messages use a message envelope for communications specifying a particular channel. A Bayeux client can authenticate, query the server capabilities, subscribe to channels, and so on. The responses from the event router and data relating to application channels are encoded using the JSON structure.

The meta channel is used for communication with the event router itself. Its channel identifier is in the form:

`/meta/<verb>`

Where verbs include the following:

- ▶ Handshake
- ▶ Connect
- ▶ Reconnect
- ▶ Disconnect
- ▶ Status
- ▶ Subscribe
- ▶ Unsubscribe
- ▶ Ping

The interactions for establishing a subscription using the Bayeux protocol are shown in Figure 10-6.

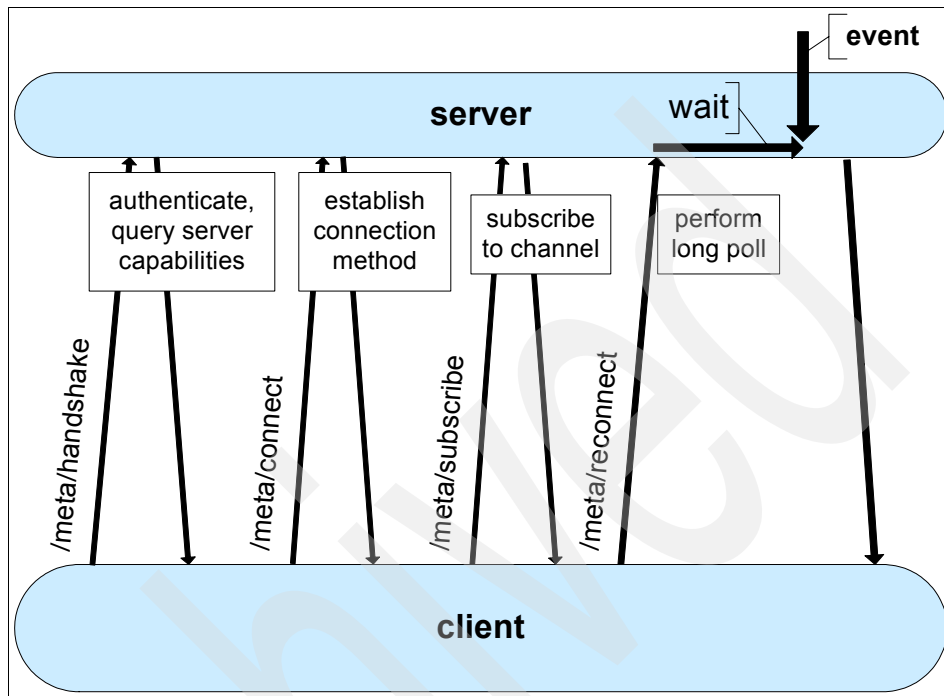


Figure 10-6 Establishing a subscription using Bayeux protocol

A sample reconnect request is shown in Example 10-1.

*Example 10-1 Reconnect request*

```
POST /quotestreamer/stockServlet HTTP/1.1
Content-Type: application/x-www-form-urlencoded
```

```
message=[{"channel":"/meta/reconnect",
"connectionType":"long-polling",
"clientId":"ci123",
"connectionId":"/meta/connections/con345",
"timestamp":"1177276704968",
"id":"9000"}]
```

The response received is shown in Example 10-2.

*Example 10-2 Reconnect response*

---

```
200 OK
Content-Language: en-GB
Content-Length: 143
Expires: Thu, 01 Dec 1994 16:00:00 GMT

[{ "id":"9001",
  "data":percentChange":-1.8,
    "gaining":false,
    "currentPrice":9820.3,
    "change":-179.7},
  "timestamp":"1177276707968",
  "channel":"IND"}]
```

---

## 10.5 Cometd and the Dojo Toolkit

Cometd (Comet Daemon) is a project to provide implementations of the Bayeux protocol on various platforms. Cometd has been developed for Perl and Java, but also as a client-side implementation as part of the Dojo Toolkit. Currently, the Dojo Toolkit (described in Chapter 9, “Ajax client runtime” on page 269) is the only JavaScript library to support the Bayeux protocol, although any JavaScript library or HTTP client that implements the Bayeux protocol support can communicate with the Web messaging service.

The Dojo toolkit cometd client module makes initialization, subscription, unsubscription, and publishing operations easy to accomplish. Through the cometd client, server-driven events are integrated into the Dojo event and topic system. See Example 10-3.

*Example 10-3 Dojo cometd client code*

---

```
<script type="text/javascript">
  dojo.require("dojox.cometd");
  dojo.addOnLoad(function() {
    dojox.cometd.init({}, "/pubsub/testServlet");
    dojox.cometd.subscribe("testtopic", false, "alertMessage");

    dojo.byId('sendBtn').onclick = function() {
      dojox.cometd.publish("testtopic", {
        'test' : "Test Comet Message"
      });
    };
  });
</script>
```

```
        return false;
    });
});

function alertMessage(message) {
    alert("Message: " + message.data.test);
}
</script>
```

---

## 10.6 Web messaging service overview

The Web message service implementation bridges incoming Bayeux requests to the service integration bus allowing Web services, JMS clients, or any item connected to the service integration bus to publish events to Web-based clients.

The Web messaging service can be used in a new or existing application by either placing a runtime Java archive (JAR) file into WebSphere Application Server, placing a utility file library JAR file in an application Web module, or setting up a simple configuration file and configuring servlet mappings. See Figure 10-7.

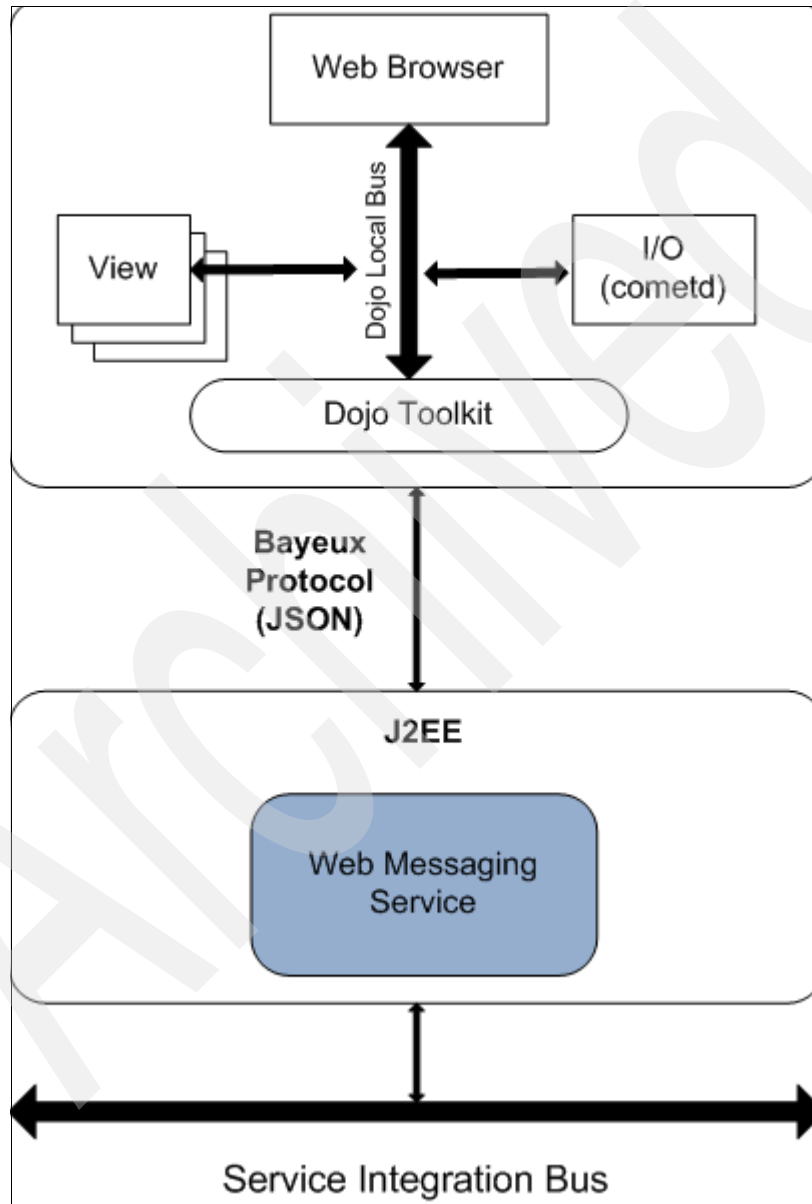


Figure 10-7 Overall Web messaging service architecture

## 10.6.1 Scaling using channels

In a typical application, a browser periodically requests updates from a server at a specified interval or polling. The Bayeux protocol differs by communicating through a long-held HTTP request in which a server typically holds a connection open for a certain time to wait for an event to push to the browser.

This use of long polling implies a large number of connections holding open connections in a pending state. With this style of communication, the Web container cannot scale, as each waiting client consumes a thread waiting for an event. Other servers that handle this style of request have different methods for scaling. See Figure 10-8.

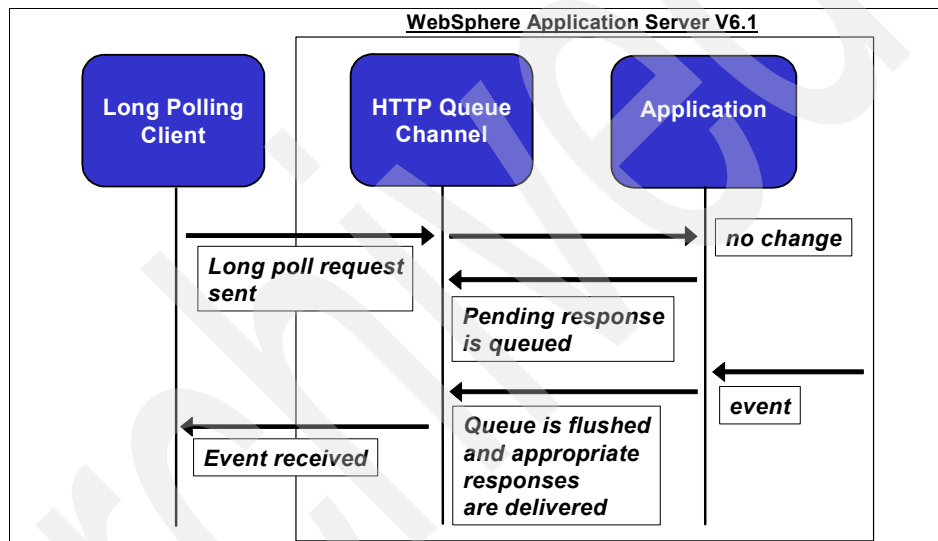


Figure 10-8 HTTP queue channel

Because the Web messaging service is designed to work with existing versions of WebSphere Application Server, and the current Web container does not have the ability to scale well with this method of communication. A new mechanism is introduced in the Web messaging service to achieve scalability.

This new mechanism takes advantage of channel framework architecture and creates WebMsg channels that extend the HTTP channel to bridge incoming Bayeux requests to the service integration bus.



## 10.6.2 Service integration bus connectivity

The Web messaging service connects browser clients to the built-in messaging engine and the service integration bus for subscribing and publishing to events and receiving messages. The underlying bridge to the service integration bus occurs through direct API calls to a service integration bus topic space. Multiple ways to publish messages to Web clients exist, since the Web clients are connected to the service integration bus. Some of these options include:

- ▶ Standard Enterprise JavaBeans™ (EJB) publishing to a topic
- ▶ A JMS client publishing to a topic
- ▶ A Web service
- ▶ The Web client itself publishing to other Web clients

Figure 10-9 shows this server architecture.

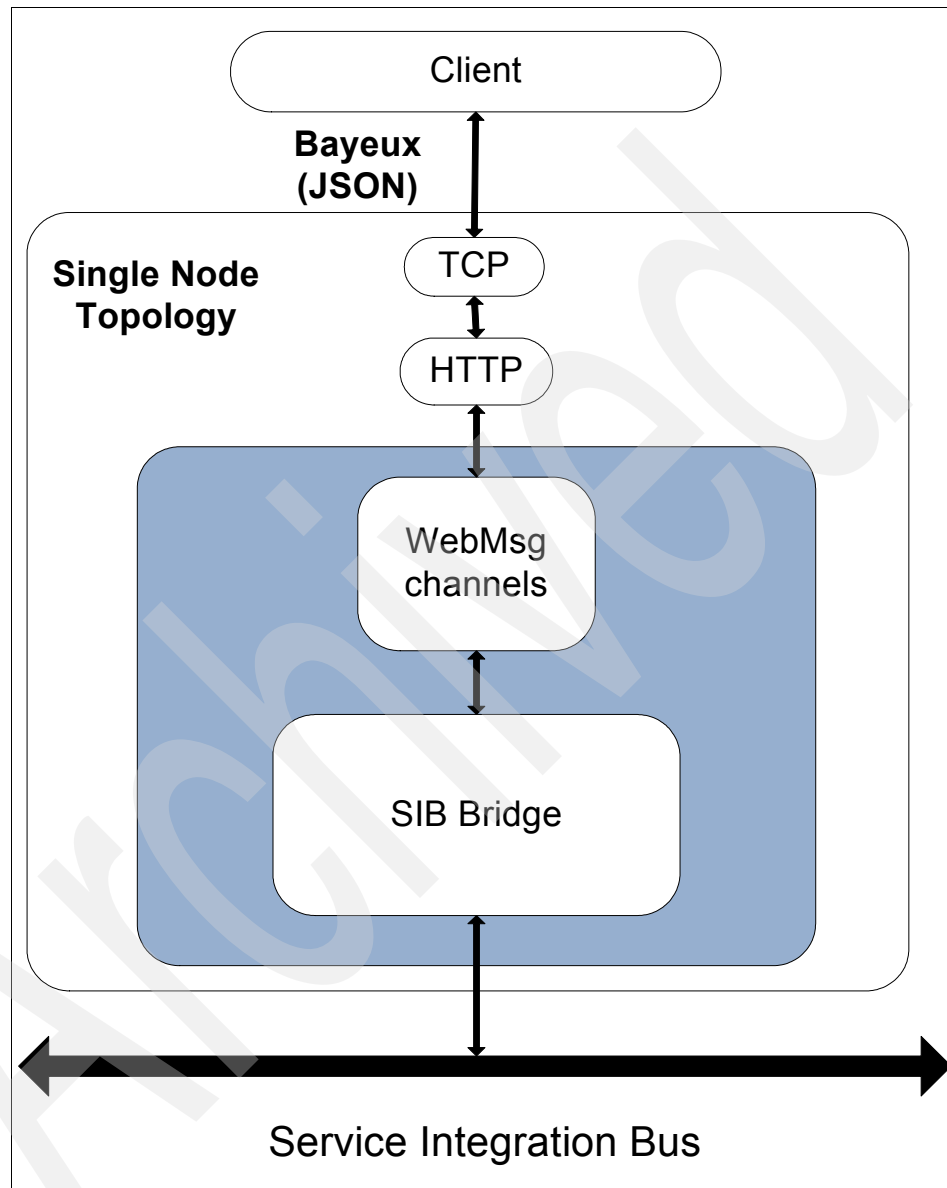


Figure 10-9 Server architecture

## 10.7 Web messaging service challenges

There are several challenges when deploying Web messaging enabled applications using the Bayeux protocol. Any Comet style application that uses long-held requests faces these challenges. Many of these difficulties deal with the strain that Comet-style communications cause on the synchronous nature of various pieces of the Internet infrastructure. The Web messaging service, through the use of the channel framework, can scale above a single thread per request, but many other pieces of the Internet infrastructure have difficulties doing so.

### 10.7.1 Web servers

One area that might have difficulty scaling is a Web server. When IBM HTTP Server for WebSphere Application Server is fronting Web messaging enabled applications, IBM HTTP Server for WebSphere Application Server needs to be configured to handle more requests than a typical application because requests spend more time waiting for an event. IBM HTTP server for WebSphere Application Server ties a single waiting request to a single thread and is constrained by the maximum number of threads available to the Web server. In most Web messaging installations, the number of IBM HTTP Server for WebSphere Application Server installations fronting a Web messaging enabled application will need to be increased.

The proxy server for WebSphere Application Server is an alternative to IBM HTTP Server for WebSphere Application Server for fronting a Web messaging enabled application. The proxy server for WebSphere Application Server does not tie a thread to each incoming request and should be able to handle a higher number of concurrent clients than IBM HTTP Server for WebSphere Application Server. Issues might arise in replacing IBM HTTP Server for WebSphere Application Server with the proxy server for WebSphere Application Server. Refer to the “Know your proxy basics”<sup>2</sup> IBM developerWorks® article for more information about fronting a Web messaging enabled application with the proxy server for WebSphere Application Server. Other hardware-based solutions might be used to front a Web messaging solution. It is important to remember that session affinity is required to bring requests back to the same server in a clustering environment.

---

<sup>2</sup> [http://www.ibm.com/developerworks/websphere/techjournal/0706\\_col\\_burckart/0706\\_col\\_burckart.html](http://www.ibm.com/developerworks/websphere/techjournal/0706_col_burckart/0706_col_burckart.html)

## 10.7.2 Two-connection limit

An Ajax Bayeux client in a Web browser will be the most common client connecting to the Web messaging service. Browsers have certain limitations when connecting to a server. One limitation is a two-connection limit to a single server. When a Bayeux connection is established, one of these connections is used, leaving one other connection free. Since one connection is used, a single browser instance, including multiple tabs and windows, will only be able to establish one Bayeux connection to a server. When a second browser window tries to establish a long-held request, this connection must be denied or reverted to traditional polling. When developing a Web messaging enabled application, you must understand and plan for this limitation.

Care must also be taken when performing other time-sensitive browser communication to the same server to which a Bayeux connection is established. The Bayeux protocol uses the second available connection to subscribe, unsubscribe, and publish information to the server. Other operations that consume a connection include:

- ▶ Ajax XMLHttpRequest operations
- ▶ Image or HTML downloading
- ▶ File uploading

When multiple operations to a server are occurring over the same browser instance, your Bayeux client application can encounter slow downs and strange behavior while waiting for a connection to be released.

## 10.8 Using the Web messaging service

Due to the nature of the features and challenges that exist as outlined in the previous topics, developing and installing a Web messaging enabled application must be carefully planned. Some items that must be planned for include:

- ▶ Understand the impacts of using long-held requests from a client to a server. Make sure that the infrastructure can support the challenges of running a Web messaging enabled application.
- ▶ Determine how a browser or client connects to the Web messaging service. See 10.12, “Web messaging service configuration” on page 482, for more information.
- ▶ The Web messaging service bridges clients to the service integration bus for publish/subscribe operations. A service integration bus must be planned, created, and configured. Refer to 10.12.1, “Service integration bus

configuration for Web messaging service” on page 483, and 10.14, “Clustered Web messaging enabled applications” on page 495, for more information.

- ▶ How messages are sent to Web messaging clients. See 10.12.3, “Web messaging service client reference” on page 492, and 10.16, “Service integration” on page 499, for more information.
- ▶ Determine key metrics, such as number of concurrent clients, number of client subscriptions, and publish rate. Configure a workload managed application infrastructure (see 10.14.2, “Workload management” on page 495) if a single server cannot handle scale to meet the required needs.
- ▶ There are special considerations when enabling security for a Web messaging enabled application. Refer to 10.15, “Securing Web messaging enabled applications” on page 496, for more information.
- ▶ Understand the concerns (see 10.14, “Clustered Web messaging enabled applications” on page 495) when deploying an application into a cluster.
- ▶ Understand what is involved in enabling an application to use the Web messaging service. Refer to 10.11, “Developing Web messaging enabled applications” on page 480, for more information.

## 10.9 Web messaging service Quick Start

The Web messaging service consists of a runtime Java archive file that gets placed in the WebSphere Application Server installation directory and a library designed to be inserted into a new or existing Web module as a utility JAR file. This section provides brief instructions for enabling the Web messaging service and developing a Web messaging enabled application. Included in this section are links to more detailed sections. In addition, the QuoteStreamer sample application can be examined and used as a starting point for developing Web messaging enabled applications.

### 1. Install the Web messaging service.

The Web messaging service must be installed to run a Web messaging enabled application. Refer to the installation instructions (4.5, “Installing Feature Pack for Web 2.0” on page 46) for more information.

2. Locate the utility library.

The Web messaging service utility library is available in the following directories:

- If the Feature Pack for Web 2.0 plug-ins have been installed into the Eclipse environment, the library file is available at:

- |       |      |
|-------|------|
| Linux | Unix |
|-------|------|

`<ECLIPSE_HOME>/plugins/com.ibm.websphere.Webmsg_1.0.0/lib/Webmsg_applib.jar`

- |         |
|---------|
| Windows |
|---------|

`<ECLIPSE_HOME>\plugins\com.ibm.websphere.Webmsg_1.0.0\lib\Webmsg_applib.jar`

- Once Feature Pack for Web 2.0 has been installed, the library file can be found in the installation tree (where `<WAS_HOME>` is the installation directory of the IBM WebSphere Application Server where Feature Pack for Web 2.0 has been installed):

- |       |      |
|-------|------|
| Linux | Unix |
|-------|------|

`<WAS_HOME>/Web2fep/optionalLibraries/MessagingService/Webmsg_applib.jar`

- |         |
|---------|
| Windows |
|---------|

`<WAS_HOME>\Web2fep\optionalLibraries\MessagingService\Webmsg_applib.jar`

3. The Web messaging service must be enabled for a Web messaging enabled application to run. Refer to 10.10, “Enabling the Web messaging service” on page 478, for more information.

- a. Log into the administrative console of WebSphere Application Server.
- b. Navigate to **Servers** → **Application Servers**.
- c. Select the current application server.
- d. Expand **Web Container Settings** and select **Web container transport chains**.
- e. Select **WCInboundDefault transport chain** or select a transport chain that corresponds to the virtual host that will be selected during application install.
- f. Select **Web Container Inbound Channel**.
- g. Select **Custom Properties**.
- h. Click **New**.
- i. Enter `Webmsgenabled` for the name property and `true` for the value.

- j. Click **Apply**.
- k. Click **Save** to save the repository information.
- l. Restart the application server now or wait until after the Configure Service Integration Bus has been set up.

4. Configure a service integration bus.

A Web messaging application must have a configured service integration bus with a defined topic space to run. At a minimum, the bus name must be provided in a Web messaging configuration file for a Web messaging application to function.

5. Insert the Dojo Toolkit into the Web module.

The Dojo Toolkit provides client support for the Web messaging service. The easiest way is to include the Dojo Toolkit in a Web module. If the Dojo Toolkit is not included in a Web module (statically served through Web server, for example), the Uniform Resource Identifier (URI) location of the `dojo.js` file must be known to load the dojo toolkit in any Web module presentation files.

6. Embed the Web messaging utility library in the Web module.

The best place to insert the Web messaging utility library is in a Web module's `WEB-INF/lib` directory as a utility library. Copy the `Webmsg_applib.jar` file located in step 2 into the Web module's `WEB-INF/lib` directory. The Web messaging utility library can be used in the same Web module as the Dojo Toolkit or in a different Web module. The key interaction point is the Bayeux protocol communication URI.

7. Determine the Web messaging service communication URI and map the Web messaging servlet to that URI.

The `"/WebmsgServlet"` servlet-mapping parameter defines the communication URI between the Dojo Bayeux client and the Web messaging service. On the server side, a servlet mapping must be created that matches the `dojox.cometd.init` initialization parameter. The servlet mapping maps to the servlet class `com.ibm.websphere.Webmsg.servlet.WebMsgServlet`. This servlet is provided in the Web messaging utility library. For more information see 10.11, "Developing Web messaging enabled applications" on page 480. A sample `Web.xml` file is shown in Example 10-4.

*Example 10-4 Web.xml snippet*

---

```
<servlet>
  <description/>
  <display-name>
    WebMsgServlet
  </display-name>
  <servlet-name>WebMsgServlet</servlet-name>
```

```
<servlet-class>
    com.ibm.websphere.Webmsg.servlet.WebMsgServlet
</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>WebMsgServlet</servlet-name>
    <url-pattern>/WebmsgServlet</url-pattern>
</servlet-mapping>
```

---

The URI defined above is used by the Dojo Toolkit cometd client for initialization purposes. A simple initialization statement in a presentation file is used and an example is shown here for comparison:

```
dojox.cometd.init("WebmsgServlet");
```

#### 8. Create a Web messaging configuration file.

The Web messaging service reads configuration parameters from a configuration file to handle incoming Web messaging requests. Create a Web messaging configuration file, `Webmsg.json`, in the `WEB-INF` directory (same directory as the `Web.xml` file), similar to Example 10-5. The `Webmsg.json` file is specified as a JSON object with each key matching an existing servlet name mapped to the Web messaging servlet. See 10.12, “Web messaging service configuration” on page 482, for more information about the configuration options.

*Example 10-5 Webmsg.json messaging configuration file*

---

```
{
  "WebMsgServlet":
  {
    "busName": "thebus",
    "destination": "Default.Topic.Space",
    "clientCanPublish": true,
    "longPollTimeout": 30
  }
}
```

---



## 9. Create the client application.

Server-side setup is now complete except for installing the application. The remaining task is to create the client-side logic using the Dojo Toolkit functions. A simple test example is shown in Example 10-6. Note how the initialization URI parameter matches the Web messaging servlet mapping.

*Example 10-6 Client code*

---

```
<html>
  <head>
    <title>cometd client/server test page</title>
    <script type="text/javascript" src="dojo/dojo.js"
      djConfig="isDebug: true, parseOnLoad: true,
      debugAtAllCosts: false">
    </script>

    <script type="text/javascript">
      dojo.require("dojox.cometd");
      dojo.addOnLoad(function(){

        dojox.cometd.init("WebmsgServlet")
        dojox.cometd.subscribe("/testtopic", window, "display");
      });

      function display(msg) {
        alert("Message " + msg.data.test);
      }
    </script>

    <div align="left">
      <button Id="publishButton"
onclick="dojox.cometd.publish('/testtopic', {test: 'Data'})";>
        <span style="font-size:small">PUBLISH</span>
      </button>
    </div>
  </head>
  <body></body>
</html>
```

---

## 10. Install and run the application.

Assemble, install, and test the application using standard Web application assemble, installation, and test procedures.

## 10.10 Enabling the Web messaging service

The Web messaging service must be enabled for a Web messaging enabled application to run. Due to the duration that Web messaging requests stay open, the existing Web container is not able to scale to handle more concurrent clients than the Web container thread pool. To solve this problem, the Web messaging service provides a new channel framework channel—a chain that shares ports with existing Web container transport chains. By default, the Web messaging transport chain is not enabled. To start the Web messaging transport chains, a custom property must be set on a Web container channel in an existing transport chain. The Web container transport chains that need to be configured to run Web messaging requests before enabling this custom property must be identified.

There are two default Web container transport chains that serve Web traffic. These two transport chains are named `WCInboundDefault` and `WCInboundSecure`. In most cases you should determine whether you will be running Web messaging requests over HTTP or HTTPS. If you are running requests over HTTP only, follow the steps below to enable the Web messaging service on the `WCInboundDefault` transport chain. If you require a secure connection over HTTPS, follow the steps below for the `WCInboundDefaultSecure` transport chain. If Web message traffic is coming in on both HTTP and HTTPS, enable the Web messaging service on both `WCInboundDefault` and `WCInboundDefaultSecure` chains. If you end up configuring a transport chain and not handling Web messaging requests through that transport chain, there is a very small amount of overhead to determine whether the Web messaging channel is involved in the request.

### Enabling the Web messaging service on distributed platforms

To enable:

1. Verify that the Feature Pack for Web 2.0 is installed. See 4.5, “Installing Feature Pack for Web 2.0” on page 46, for more information.
2. Log into the administrative console of the application server.
3. Navigate to **Servers** → **Application Servers**.
4. Select the current application server.
5. Expand **Web Container Settings** and select **Web container transport chains**.
6. Select the transport chain identified above.
7. Select **Web Container Inbound Channel**.
8. Select **Custom Properties**.
9. Click **New**.

10. Enter `Webmsgenabled` for the name property and `true` for the value.
11. Click **Apply**.
12. Click **Save** to save the repository information.
13. Restart the server for the changes to take effect.

## Enabling the Web messaging service on z/OS

After completing the previous steps, follow the steps below to complete Web messaging enablement on z/OS.

1. Verify that the Feature Pack for Web 2.0 is installed. See 4.5, “Installing Feature Pack for Web 2.0” on page 46, for more information.
2. Log into the administrative console of WebSphere Application Server.
3. Navigate to **Servers** → **Application Servers**.
4. Select the current application server.
5. Expand **Web Container Settings** and select **Web container transport chains**.
6. Select the transport chain identified above.
7. Select **HTTP Inbound Channel**.
8. Set the discrimination weight to 1.
9. Click **Apply**.
10. Click **Save** to save the repository information.
11. Restart the application server for the changes to take effect.

It is possible that additional Web container transport chains have been enabled to serve Web traffic. To determine whether the Web messaging service should be enabled for these transport chains, determine which Virtual hosts your Web messaging enabled application will be installed to, then enable the corresponding transport chains that match the virtual host's ports.

In some situations, it might be desirable to create a new Web container transport chain for the sole purpose of handling Web messaging requests. Refer to the “Know your proxy basics”<sup>3</sup> IBM developerWorks article. The following section provides information about creating a new Web container transport chain with its own thread pool.

---

<sup>3</sup> [http://www.ibm.com/developerworks/websphere/techjournal/0706\\_col\\_burckart/0706\\_col\\_burckart.html](http://www.ibm.com/developerworks/websphere/techjournal/0706_col_burckart/0706_col_burckart.html)

## 10.11 Developing Web messaging enabled applications

A new or existing Web application can be enabled to use the Web messaging service. Enabling a Web messaging application consists of creating or modifying a Web module to add the necessary Web messaging function. Enabling the Web messaging server-side function is a configuration-oriented task. The Web messaging service handles all details of communicating to Web messaging clients.

The application developer and deployer need to understand the best way to configure the Web messaging service. When creating a Web messaging enabled application, an application or Web developer needs to write the client logic to connect to the Web messaging service to receive messages delivered by the server. The target client type for the Web messaging service is an Asynchronous JavaScript and XML client, and an application or Web developer needs Ajax toolkit skills to create the client-side logic.

Once a Web messaging enabled application is created, the Web messaging enabled application is deployed and managed using standard Web application techniques. For step-by-step instruction on how to enable an application for Web messaging usage, refer to 10.9, “Web messaging service Quick Start” on page 473. Refer to the QuoteStreamer sample application for a reference Web messaging enabled application.

### 10.11.1 Web messaging utility library

A J2EE utility library is provided for inclusion into your Web module. This library is named `Webmsg_applib.jar` and must be included in any Web messaging enabled application. This JAR file is typically placed in your Web module's `WEB-INF/lib` directory for application usage. Included in this utility library is a Web messaging servlet that must be defined and used to create servlet mappings and an application-level publishing library to easily publish to Web messaging clients through the service integration bus.

### 10.11.2 Web messaging servlet

At least one servlet mapping must be defined in your Web messaging enabled application. Web messaging clients connect to these mappings for Web messaging operations. Included in the Web messaging utility library is an existing servlet with the class name of `com.ibm.websphere.Webmsg.WebMsgServlet`. One or more definitions to this existing servlet should be defined, mappings to these definitions should be created, and a corresponding entry in the Web messaging configuration with the servlet name must be created. Refer to 10.9,

“Web messaging service Quick Start” on page 473, to see an example of a step-by-step enablement.

The Web messaging servlet provided works in conjunction with the underlying Web messaging channel framework piece to provide Web messaging communication. Consider the following points when defining new Web messaging servlets and servlet mappings:

- ▶ The Web messaging service uses the Bayeux protocol for communication. As part of client-to-server communication through the Bayeux protocol, a new Bayeux session is established through one or more handshake messages. The Web messaging servlet processes the handshake request only, leaving subsequent Bayeux operations, such as connect and subscribe, to be handled by the Web messaging runtime component, which bridges these operations to the service integration bus. Standard J2EE authentication and authorization methods can be used to protect access to the incoming URI. Refer to 10.15, “Securing Web messaging enabled applications” on page 496, for more information.
- ▶ Client communication to the Web messaging service requires session affinity. The Web messaging servlet establishes a new session through the use of getSession upon a successful Bayeux handshake request. The Web messaging service does not store or retrieve information from an HTTP session. Refer to 10.14, “Clustered Web messaging enabled applications” on page 495, for more information.
- ▶ Multiple mappings to the same Web messaging defined servlet each use the same Web messaging configuration options. A new Web messaging servlet definition and entry in the Web messaging configuration file must be created to vary configuration options for a different incoming URI.

## 10.12 Web messaging service configuration

The Web messaging service reads configuration information from the file, `Webmsg.json`, located in a Java 2 Platform, Enterprise Edition (J2EE) application Web module `WebContent/WEB-INF` directory. This information is read on application start and stop to initialize the Web messaging service with configuration information. This file resides alongside the `Web.xml` file of a Web module and the location of this file cannot be changed. The `Webmsg.json` file is specified in JavaScript Object Notation (JSON) format. The contents of the file should represent a JSON object. A JSON object is a collection of key/value pairs. A Web messaging configuration file is shown in Example 10-7.

*Example 10-7 Webmsg.json messaging configuration file*

---

```
{
  "WebMsgServlet":
  {
    "busName": "thebus",
    "destination": "Default.Topic.Space",
    "clientCanPublish": true,
    "longPollTimeout": 30
  }
}
```

---

Each key in the JSON object specifies an existing servlet name in the same Web module where the `Webmsg.json` file is specified. The value for each key is another JSON object where Web messaging configuration options are specified. Incoming Web messaging requests are determined by the servlet mappings for the specified servlet. These incoming requests are processed with the specified configuration options. Example 10-8 is a snippet of a `Web.xml` file containing the `WebMsgServlet` servlet and associated servlet mappings.

*Example 10-8 Web.xml snippet*

---

```
<servlet>
  <description/>
  <display-name>
    WebMsgServlet
  </display-name>
  <servlet-name>WebMsgServlet</servlet-name>
  <servlet-class>
    com.ibm.websphere.Webmsg.servlet.WebMsgServlet
  </servlet-class>
</servlet>
```

```
<servlet-mapping>
  <servlet-name>WebMsgServlet</servlet-name>
  <url-pattern>/stockServlet</url-pattern>
</servlet-mapping>
```

---

In Example 10-8 on page 482, the WebMsgServlet servlet is mapped to /stockServlet. Any incoming Web messaging requests that map to /stockServlet are processed with the configuration parameters in the Webmsg.json shown Example 10-7 on page 482. In this case, the Web messaging requests use service integration bus “thebus”, the default topic space “Default.Topic.Space”, and the Bayeux long poll time out of 30, and Web messaging clients can publish messages to the server.

Most configurations only need to create one servlet and the related Webmsg.json configuration file. There are times when multiple definitions must be created in a single Web module. Examples include connecting to different service integration buses, specifying a different default destination, or separating clients by Bayeux long poll time out. To separate incoming requests based on configuration parameters, a new servlet definition is mapped to com.ibm.websphere.Webmsg.servlet.WebMsgServlet and a new servlet mapping must be created. Finally, create a new entry in the Webmsg.json with the desired configuration parameters. This entry must have a key that maps to the newly created servlet.

### 10.12.1 Service integration bus configuration for Web messaging service

The Web messaging service makes use of the service integration bus provided as part of WebSphere Application Server V6.0.x and later to provide the underlying publish and subscribe capability required to support Bayeaux clients. To determine how Bayeaux publishers and subscribers interact with the service integration bus, it is necessary to provide some configuration parameters when deploying an application that provides Bayeaux capability.

The Bayeaux subscription channel concept is equivalent to a topic in the service integration bus such that it enables classification of events into related groups, for example, /sports/football or /weather. In the service integration bus, topics exist within a topic space destination to which producers and consumers attach. Topics in one topic space are completely independent of topics, even potentially with the same name, that exist in a different topic space. Part of the configuration of the Feature Pack for Web 2.0 thus gives the application developer the ability to specify which service integration bus topic space is used by Bayeaux applications.

Additionally, events (messages) published to a service integration bus topic space have a reliability applied to them that determines how the messages are treated inside the bus. This includes the conditions under which the messages can be discarded by the messaging infrastructure. In some scenarios, it is acceptable for messages to be discarded if the server becomes resource constrained, possibly due to a high workload, while in others it is preferable to retain messages even at the risk of overloading the server.

When configuring advanced scenarios that include concepts such as workload management or security it is also necessary to consider the implications of those requirements on the service integration bus. Each of these concepts is described in further detail below.

The options discussed in the following sections are available for configuring the way in which Bayeaux applications interact with the service integration bus.

## busName

A service integration bus is required for Web messaging operations. Clients can only connect to a single service integration bus through a single incoming URI. Connecting to multiple buses is only possible by creating a separate Web messaging configuration definition and specifying a different bus name, but the client will have to connect through separate URIs and use multiple connections. When using an Ajax client it is only feasible to make one Bayeux connection to a single server on a single Web page. Different pages can specify different URIs and connect to different buses. Table 10-1 shows the SIBus property options.

Table 10-1 *busName SIBus property options*

<b>Name</b>	busName
<b>Required/optional</b>	Required
<b>Data type</b>	String
<b>Supported values</b>	The name of any service integration bus
<b>Default value</b>	No default
<b>Description</b>	The service integration bus that incoming Web messaging clients use for publish or subscribe operations



## destination

In the majority of scenarios a Bayeaux application needs to consume or produce messages from a single service integration bus topic space. The destination property enables the application developer to specify which topic space is used for all interactions with the service integration bus.

Table 10-2 destination SIBus property options

<b>Name</b>	Destination.
<b>Required/optional</b>	Optional.
<b>Data type</b>	String.
<b>Supported values</b>	The name of any service integration bus topic space.
<b>Default value</b>	Default.Topic.Space.
<b>Description</b>	<p>This is the topic space on the service integration bus that the application consumes from or produces to.</p> <p>The destination specified can be a real topic space, or alternatively it can be an alias (see “Alias destinations” in the Redbooks publication <i>Enabling SOA Using WebSphere Messaging</i>, SG24-7163) to a topic space. Using an alias enables the application to be isolated from the actual name of the topic space.</p> <p>When using this property to nominate the topic space that is used by Bayeaux applications, the name of the topic inside the topic space is determined by taking the Bayeaux subscription channel name and removing the leading slash. For example, a channel name of “/sports/football” maps to a service integration bus topic of “sports/football”.</p> <p>If the application developer is not aware of which topic space is used, then this property can be omitted and the default topic space is used.</p>

## multipleDestinations

In advanced scenarios it might be necessary for a Bayeaux application to produce or consume more than one service integration bus topic space, for example, to display data from a range of disparate sources as part of a portlet environment. In this case, the application developer can enable the `multipleDestinations` property, which overrides the `destination` property and enables the developer to specify the name of the required topic space (or alias) destination as part of the Bayeaux subscription channel name.

Table 10-3 *multipleDestinations SIBus property options*

<b>Name</b>	<code>multipleDestinations</code>
<b>Required/optional</b>	Optional
<b>Data type</b>	Boolean
<b>Supported values</b>	True or false
<b>Default value</b>	False
<b>Description</b>	<p>Enables an alternate Bayeaux subscription channel syntax that supports specifying the name of the target service integration bus topic space as part of the channel name.</p> <p>When enabled, this property causes the first channel segment that is used to specify the name of the topic space. For example, to publish to a topic "track/relay4x100" in the topic space "OlympicResults", the application specifies the following Bayeaux subscription channel:</p> <pre>/OlympicResults/track/relay4x100</pre> <p>Note that if <code>multipleDestinations</code> is enabled, then the value of the 'destination' property is ignored.</p>

## reliability

Events (messages) published to a service integration bus topic space have a reliability applied to them that determines how the messages are treated inside the bus. This includes the conditions under which the messages can be discarded by the messaging infrastructure. In some scenarios, it is acceptable for messages to be discarded if the server becomes resource constrained (possibly due to a high workload), while in others it is preferable to retain messages even at the risk of overloading the server.

Table 10-4 *reliability SIBus property options*

<b>Name</b>	Reliability.
<b>Required/optional</b>	Optional.
<b>Data type</b>	String.
<b>Supported values</b>	<ul style="list-style-type: none"><li>▶ BestEffortNonPersistent</li><li>▶ ExpressNonPersistent</li><li>▶ ReliableNonPersistent</li><li>▶ ReliablePersistent</li><li>▶ AssuredPersistent</li></ul>
<b>Default value</b>	ExpressNonPersistent.
<b>Description</b>	<p>This specifies the level of reliability that events (messages) published by Bayeaux applications are sent to the service integration bus.</p> <p>Details on the behavior of these options can be found in Redbooks publication <i>Patterns: Implementing Self-Service in an SOA Environment</i>, SG24-6680.</p> <p>Note that for non-durable subscriptions, messages are not available following a server restart because the subscription no longer exists. This means that there is limited benefit in choosing one of the two persistent reliability levels. In most cases, one of the non-persistent values should be used.</p>

## clientCanPublish

The Bayeux protocol supports clients publishing to the service integration bus. This can be enabled by setting the clientCanPublish Bayeux configuration option to true. To have greater control over inspecting the clients' published message contents, you might, for example, prefer to require clients to publish through a Web application (for example, a servlet) before publishing the message to the service integration bus. By default, clients cannot publish through the Bayeux protocol.

Table 10-5 *clientCanPublish SIBus property options*

<b>Name</b>	clientCanPublish.
<b>Required/optional</b>	Optional.
<b>Data type</b>	Boolean.
<b>Supported values</b>	True or false.
<b>Default value</b>	False.
<b>Description</b>	Indicates whether Web messaging clients are permitted to publish. By default, clients cannot publish. You must explicitly set clientCanPublish to true to support client publishing.

## authType

Incoming Web messaging clients must be authenticated and authorized to connect to the service integration bus when bus security is enabled. Refer to 10.15, “Securing Web messaging enabled applications” on page 496, for more information. Table 10-6 describes the authType SIBus property options.

Table 10-6 *authType SIBus property options*

<b>Name</b>	authType.
<b>Required/optional</b>	Optional.
<b>Data type</b>	String.
<b>Supported values</b>	<ul style="list-style-type: none"><li>▶ none</li><li>▶ auth_alias</li><li>▶ basic</li><li>▶ sso</li></ul>
<b>Default value</b>	None.

<b>Description</b>	<p>When bus security is enabled, incoming Web clients need an identity to authorize to the service integration bus. Specify this configuration parameter to determine where authentication information is obtained to log in to the service integration bus.</p> <ul style="list-style-type: none"> <li>▶ The none option indicates that no user ID and password information is used to log in to the service integration bus.</li> <li>▶ The auth_alias option indicates that the user ID and password obtained from the authAlias configuration directive are used to log in to the service integration bus.</li> <li>▶ The basic option can be used when Web security is enabled for the application. The basic option uses the credentials from the basic authentication header to log in to the service integration bus.</li> <li>▶ The sso option can be used when Web security is enabled for the application. The sso option uses the credentials from LTPA cookies to log in to the service integration bus.</li> </ul> <p>If any of the options fail to retrieve credentials, a login to the service integration bus with a blank user ID and password is performed. If service integration bus security is enabled, Web clients are not able to connect for Web messaging operations.</p>
--------------------	--

## authAlias

The authAlias configuration property specifies an existing Java 2 Connector authentication alias that incoming Web messaging clients will use to access the service integration bus. Table 10-7 describes the authAlias SIBus property options.

Table 10-7 authAlias SIBus property options

<b>Name</b>	authAlias.
<b>Required/optional</b>	Optional.
<b>Data type</b>	String.
<b>Supported values</b>	A valid J2C authentication alias.
<b>Default value</b>	None.

**Description:**

When bus security is enabled, incoming Web clients need an identity to authorize to the service integration bus. Specify this configuration parameter to give all incoming Web clients access to the service integration bus with the user ID and password indicated by the authentication alias. Only valid when authType is set to auth\_alias.

## 10.12.2 Bayeux configuration

The Web messaging service uses the Bayeux protocol for publish/subscribe communication. Only the Bayeux long-polling transport option is supported. Several aspects of the Bayeux communication are configured in a Web messaging configuration file. These configuration options are listed below.

### **longPollTimeout**

In the Bayeux protocol, a server does not complete the HTTP request until messages are available. The longPollTimeout configuration option specifies how long the server keeps the request open before completing the request. When the request is completed, a Bayeux client is immediately re-connected to the server unless the clientPollInterval is set. Care should be taken when setting these values. You must set the clientPollInterval value to a reasonable value if the longPollTimeout value is set to 0 to avoid flooding the server with repeated Bayeux connect requests.

Table 10-8 describes the longPollTimeout Bayeux property option.

*Table 10-8 longPollTimeout Bayeux property options*

<b>Name</b>	longPollTimeout.
<b>Required/optional</b>	Optional.
<b>Data type</b>	Integer.
<b>Supported values</b>	Value specified in seconds between 0 and 300.
<b>Default value</b>	30.
<b>Description</b>	Indicates how long the server keeps the HTTP request open to wait for messages. Long polling, by definition, requires requests to be completed in a reasonable amount of time. The clients typically reconnect immediately upon request closure.

## clientPollInterval

The clientPollInterval specifies how long the client waits between Bayeux connect requests. If the clientPollInterval is set, the Bayeux advice mechanism is used to direct clients on how frequently they reconnect. Setting longPollTimeout to 0 and clientPollInterval to a positive value indicates that traditional polling will be used to connect to the server. Depending on the application type you are developing, traditional polling might be a preferable way to connect to the server. Configuring traditional polling allows easy connectivity to the service integration bus for message delivery. Table 10-9 describes the clientPollInterval Bayeux property option.

*Table 10-9 clientPollInterval Bayeux property options*

<b>Name</b>	clientPollTimeout.
<b>Required/optional</b>	Optional.
<b>Data type</b>	Integer.
<b>Supported values</b>	Value in seconds that is greater than or equal to 0. The value must be less than bayeuxSessionTimeout.
<b>Default value</b>	0.
<b>Description</b>	Directs clients to reconnect at specified interval. Uses the Bayeux reconnect interval advice mechanism to direct clients on how often to reconnect. This client connecting must support the Bayeux reconnect interval advice mechanism for this function to work. Clients might also connect at their own intervals.

## bayeuxSessionTimeout

The bayeuxSessionTimeout indicates how long the server waits for a client to reconnect before discarding any client session information. The client explicitly calls the Bayeux disconnect operation, but there is no guarantee that the client will call disconnect. The default value of 90 seconds reflects the fact that Bayeux sessions are meant to be short lived. This value should be set sufficiently higher than the clientPollInterval to ensure that clients can reconnect successfully and continue receiving messages. Table 10-10 describes the bayeuxSessionTimeout Bayeux property option.

Table 10-10 bayeuxSessionTimeout Bayeux property option

<b>Name</b>	bayeuxSessionTimeout.
<b>Required/optional</b>	Optional.
<b>Data type</b>	Integer.
<b>Supported values</b>	Value in seconds greater than 10.
<b>Default value</b>	90.
<b>Description</b>	Specifies how long the server waits for a client to reconnect before discarding knowledge of the client. If a client reconnects with the same client ID after the Bayeux session has expired, the Bayeux rehandshake advice mechanism is used to initiate a new Bayeux session.

### 10.12.3 Web messaging service client reference

The Web messaging service uses the Bayeux protocol for client/server publish/subscribe operations. There are many benefits to using the Bayeux protocol communication including:

- ▶ **Client support:** Any client that supports the Bayeux protocol is able to connect to the Web messaging service.
- ▶ **Application compatibility:** Any application that uses the Bayeux protocol can be transferred between Bayeux server implementations with little or no changes.
- ▶ **Protocol extensibility:** The Bayeux protocol was designed to easily be extended to support advanced functions.

A Web browser using Ajax communication is the intended client for using the Web messaging service. The Dojo toolkit includes a Bayeux client for communicating to a Bayeux-enabled server. If your Ajax toolkit of choice does not include Bayeux support, it is possible to write your own JavaScript Bayeux



client by following the Bayeux protocol spec and implementing the required client logic.

Even though an Ajax client is the intended client for use with the Web messaging service, any other client that supports the Bayeux protocol might work. When enabling Web messaging application security or deploying a Web messaging application to a cluster, you must make sure that the client supports the necessary security and session affinity function to successfully operate. Many clients are currently in development and should work with the Web messaging service as long as they are compatible with the Bayeux protocol and support the Web messaging service security and clustering features if enabled.

## 10.13 Publishing messages to the Web messaging service

When an application is configured to use the Web messaging service, there are multiple ways to publish information to browser clients, as we discuss in this section.

### 10.13.1 Browser client publishing

The first method is client publishing through the browser. Publishing is part of the Bayeux protocol. Publishing from the browser enables a different style of Web application. Some examples can be embedding chat and presence into a Web page, a collaborative Web editing application, or game application. The `clientCanPublish` option must be set to `true` in the Web messaging configuration file to allow client publishing. The code to publish within the browser is similar to the following Dojo Toolkit cometd client example:

```
dojo.cometd.publish("/testtopic", { test: "data" });
```

### 10.13.2 Publishing through a Web application

The second method is publishing to Web messaging clients through a Java 2 Platform, Enterprise Edition (J2EE) application. The Web messaging service bridges clients to the service integration bus for message publishing and delivery. Messages that need to be sent to Web clients must be routed through a service integration bus. Standard J2EE service integration bus publishing methods should be used. These standard methods can be used in an enterprise bean or servlet. Refer to 10.16, “Service integration” on page 499, for additional information.

To assist in publishing to Web messaging clients, a publishing API is provided in the Web messaging application utility library. Publishing to a Web messaging client is simplified when using the publishing API as opposed to standard JMS publishing methods. These simplifications include Bayeux channel-to-service integration topic mapping and easy identification of supported JMS message types. Use the following steps when using the publishing API:

1. Create a topic connection factory to connect to a service integration bus. Note the Java Naming and Directory Interface (JNDI) name for the topic connection factory.
2. Create a reference to PublisherServlet and pass the JNDI name in for the connection factory as shown in Example 10-9.

---

*Example 10-9 Reference to PublisherServlet*

---

```
<servlet>
  <display-name>Publisher</display-name>
  <servlet-name>Publisher</servlet-name>
  <servlet-class>
    com.ibm.websphere.Webmsg.publisher.jndijms.JmsPublisherServlet
  </servlet-class>
  <init-param>
    <param-name>CONNECTION_FACTORY_JNDI_NAME</param-name>
    <param-value>java:comp/env/jms/QuotePublish</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
```

---

3. Obtain an instance of Publisher, create a Web message, and publish as shown in Example 10-10.

---

*Example 10-10 Publishing a Web message*

---

```
Publisher publisher =
(Publisher)servletContext.getAttribute(JmsPublisherServlet.PUBLISHER
_SERVLET_CONTEXT_KEY);
if (publisher == null) {
  throw new ServletException("Publisher servlet not initialized");
}

try {
  publisher.publish(new BayeuxJmsTextMsg(s.getTopic(), data));
} catch (PublisherException e) {
  logger.log(Level.WARNING, "Error publishing data.");
}
```

---

Read the publishing API<sup>4</sup> documentation for the Web messaging publishing API.

## 10.14 Clustered Web messaging enabled applications

You can install a Web messaging enabled application to a cluster like a standard Java 2 Platform, Enterprise Edition (J2EE) application. You can cluster a Web messaging enabled application to take advantage of workload management and some high-availability capabilities.

### 10.14.1 Application install

Before you install a Web messaging enabled application, verify that each cluster member has the Web messaging service installed and enabled. If a Web messaging enabled application is installed to a server or cluster member without the Web messaging service installed or enabled, incoming Web messaging requests will fail. As a result, errors are issued to the error log. You must also make sure that the Web messaging service is installed and enabled if you are adding a new cluster member to a cluster in which a Web messaging application is installed. See 10.18, “Troubleshooting the Web messaging service” on page 500, for more error information about installation and enablement errors.

### 10.14.2 Workload management

Installing a Web messaging application to a cluster allows the application to scale beyond the means of a single application server. You should follow the same considerations when installing a Web messaging enabled application to a cluster compared to a regular application. Session affinity is required for Web messaging applications in a clustered environment, and requests must be able to be routed back to the same server, either through the Web server plug-in or the proxy server. Refer to the Redbooks publication *WebSphere Application Server Network Deployment V6: High Availability Solutions*, SG24-6688, for more information.

When deploying a Web messaging application to a clustered environment it is also necessary to consider the service integration bus implications of running in a cluster. At a high level, making a cluster into a bus member of a service integration bus makes it possible to create a set of logically equivalent messaging engines that service workload directed to the cluster.

---

<sup>4</sup> <http://publib.boulder.ibm.com/infocenter/wasinfo/v6r1/index.jsp?topic=/com.ibm.websphere.Webmsg.help/javadoc/index.html>

For cluster bus members hosting Web messaging applications, you can configure a *scalability* style messaging engine configuration to create one messaging engine for each server in the cluster. The scalability style messaging engine configuration also sets policies on each messaging engine so that each engine has a preference for one of the servers in the cluster.

This arrangement means that when Web messaging requests are routed to one of the servers in the cluster there is always a local messaging engine for that server that is capable of handling the necessary messaging actions without communicating with a different server.

The HTTP session affinity mechanism means that once a stateful relationship has been established between a Web messaging client (for example, a Web browser) and a specific server in the cluster, subsequent requests by that application are routed back to the same server that the application initially communicated with.

## 10.15 Securing Web messaging enabled applications

There are two different aspects to configuring a Web messaging enabled application:

- ▶ Protecting the incoming URI
- ▶ Configuring client authentication and authorization to the service integration bus

### 10.15.1 Protecting a Web messaging URI

Depending on the type of Web messaging application that you are writing, you might want to restrict access to a Web messaging URI. Web messaging URIs can be protected using standard Web application security methods. When a Web messaging URI is protected, the Bayeux handshake request will be authenticated and authorized using normal Web application security methods.

When application security is enabled, it is typical to enable transport security. The Web messaging service shares Web container transport chains, and if you enable the Web messaging service on a secured transport chain, communication will be secured for all Bayeux operations.

## 10.15.2 Service integration bus authentication and authorization

When configuring a Web messaging enabled application with security enabled it is important to understand that the security requirements should be extended to the service integration bus interactions unless there is a specific reason for not doing so. When planning your security model you should consider each of the aspects of security on the service integration bus described below. When Web messaging clients connect to a secured service integration bus, the client must be authenticated and authorized to access service integration bus resources.

A Web messaging client must provide credentials to authenticate to a secured service integration bus. There are two options for methods for providing credentials:

- ▶ Java 2 Connector authentication alias
- ▶ Web credentials

The best method to use depends on the needs of the application. The `authType` and `authAlias` Web messaging configuration options specify which type to use. Refer to 10.12.1, “Service integration bus configuration for Web messaging service” on page 483, for detailed configuration information. Table 10-11 describes SIBus authentication and authorization.

Table 10-11 SIBus authentication and authorization

<b>Java 2 Connector authentication alias</b>	<p>A Java 2 Connector authentication alias is one method to allow Web messaging clients to authenticate to the service integration bus. An authentication alias should be used in the following situations:</p> <ul style="list-style-type: none"><li>▶ Web security is disabled and bus security is enabled. Specifying an authentication alias is the only possible solution for allowing Web messaging clients to access the service integration bus.</li><li>▶ It is permissible for all Web messaging clients that come in on a single URI to have the same service integration bus topic space and topic permissions. In this case, it is easier to manage the authorization permissions for a single authentication instance than to manage permissions for every authenticated Web user.</li></ul>
--	--

<b>Web credentials</b>	<p>When Web security is enabled, Web credentials can be used as authentication information for connecting to a secure service integration bus. Basic authentication and LTPA single-sign on cookies are the two supported Web credential types supported. You should set your Web messaging configuration file to use an authType of basic or sso under the following conditions:</p> <ul style="list-style-type: none"> <li>▶ Web security is enabled. Either basic auth or LTPA cookie information must be readable in the HTTP request. If this information is unavailable, then access to a secure service integration bus will fail. The ability to read Web credentials is likely limited when a trust association interceptor is in use. In that case you will be limited to using Java 2 Connector authentication alias.</li> <li>▶ Fine-grained Web messaging user topic space or topic authorizations are required. In this case, you will have to follow the authorization steps below to allow user or group access to service integration bus resources.</li> </ul>
------------------------	--

The service integration bus provides a series of security checks that allow authorization to be validated at a variety of granularities. The authorization levels shown in Table 10-12 must be met for all authentication data specified.

Table 10-12 SIBus authorization levels

<b>Bus connector authorization</b>	The first level of authorization required in order to access a service integration bus is the ability to connect to the bus, or the bus connector authorization. Like other authorizations it is possible to grant this access to individual users or to a group of users.
<b>Topic space (destination) security</b>	<p>Once an application has passed the bus connector authorization check the next thing it will need to do is access a destination. In the case of Bayeaux applications this will be the topic space destinations specified in the Bayeaux configuration or channel name.</p> <p>Each destination has a set of individual roles associated with it that control the exact type of action that a particular application or user can carry out. For Bayeaux applications the most common roles of interest are the receiver role to enable the application to create subscriptions and receive events and the sender role for publishing events to the topic space.</p>
<b>Topic security</b>	Within a topic space it is also possible to configure fine-grained access control over individual topics or branches of topics. This means that one application can be prevented from seeing the events published on other topics by a different application, even if those events are being published in the same topic space.

## 10.16 Service integration

In this section we discuss service integration.

### 10.16.1 Bayeux channel-to-service integration bus channel mapping

The Bayeux channel format is different from the service integration bus topic format. A Bayeux channel is always preceded by a slash (/), while the service integration bus topic is never preceded by a slash. When publishing from a Bayeux client or through the provided publishing API, the topic conversion between Bayeux channel format and service integration format is handled by the Web messaging function. The same is true for outgoing messages. When publishing through other JMS mechanisms and an expected target is a Web messaging client, you must be aware of the slight differences in topic format. For example, the Bayeux channel /stocks/IBM is equivalent to stocks/IBM in the service integration bus topic.

### 10.16.2 JMS object types support

A limited number of JMS message types are supported when publishing to Web messaging clients. These message types currently include JMS text and object messages. The contents of a JMS text message must be a JSON serializable value:

- ▶ JSON object
- ▶ JSON array
- ▶ String value

The contents of the text message are placed into the Bayeux event delivery message data field. A JMS Object message is also supported, but limited objects can be passed. These objects are limited to:

- ▶ JSON object
- ▶ JSON array
- ▶ Java number
- ▶ Java string
- ▶ Java boolean

## 10.17 Logging and trace for the Web messaging service

The Web messaging service uses standard WebSphere Application Server logging and trace mechanisms. The package name for Web messaging service is `com.ibm.ws.Webmsg.*` or `com.ibm.websphere.Webmsg.*`.

All configuration errors and operation errors are logged. If you encounter any errors running the Web messaging service, check the logs first for more information about the cause of the problem.

## 10.18 Troubleshooting the Web messaging service

This section contains troubleshooting information for the Web messaging service.

### 10.18.1 Runtime enablement reference

The Web messaging service run time must be installed and enabled before a Web messaging enabled application is installed. If you install a Web messaging enabled application without the run time installed and enabled, the following error message is displayed, along with a 503 servlet error, when you try to access a Web messaging URI:

```
CWPSB1701E:The Web messaging service is not configured for the
following incoming URI: /uri
```

When you encounter this message, you should first verify that the Web messaging service is installed and enabled. See 10.10, “Enabling the Web messaging service” on page 478, for more details. Also note that the appropriate iFix must be installed (see 4.3, “Installing the application server product” on page 44). Successful installation and enablement of the Web messaging service displays the following log messages on server start:

```
CWPSB1102I: The Web messaging service has started.
```

If you still encounter a configuration error after the Web messaging service is installed and enabled, verify the following:

- ▶ Did you enable the Web messaging service on the correct transport chain? Check the virtual host that the Web messaging enabled application is enabled on and verify that the associated Web container transport chain is enabled.
- ▶ Check the error logs for other configuration errors. These errors are displayed on application start. In many cases, a configuration error, such as specifying



an invalid Web messaging configuration file, will cause the above failure messages. See developing a Web messaging enabled application for more information.

### 10.18.2 Client error message reference

The Web messaging service uses the Bayeux protocol. For most Bayeux protocol message fields there is a successful boolean field and an associated error field. If a Bayeux operation is unsuccessful, the Web messaging service attempts to send error information to assist application developers and users in determining why the operation failed.

The Bayeux error string format contains three arguments separated by colons. The first argument is an error code for easy comparison purposes. The second argument is a comma-separated list of arguments that are involved in the failure. The last argument is a short description of the error message. Example error messages follow:

- ▶ "error": "405:/meta/invalidchannel: The meta channel specified is not valid."
- ▶ "error": "403:/topics/notauthorized: Access to the specified channel is forbidden."
- ▶ "error": "401:s34sd34w323asdfsdf2323: The client ID specified is unknown."

Table 10-13 lists most error messages that the Web messaging service sends to Bayeux clients when an operation fails. Refer to this table when debugging client failures. In many cases, check the server logs for additional information when encountering these client errors.

Table 10-13 Web messaging service error codes and messages

Error code	Short description	Args	Description
300	Version number mismatch.	Minimum and maximum versions passed to the server.	The server could not find a version match for the client-supplied version and minimum version fields. In most cases, the server will request that the client re-handshake with a version number as supplied by the server in the version and minimumVersion fields.

Error code	Short description	Args	Description
301	Connection type mismatch.	The connection types supplied by the client.	The client tried to handshake or connect to a connection type not supported by the server. The handshake response will contain a list of supported connection types. If this error message is received during a handshake request, the client should try to reconnect with a valid value from the list of supported connection types. If this error is given in response to a connect request, the server will direct the client to rehandshake and agree on a common connection type.
302	Extension mismatch.	The extension provided.	The Bayeux protocol provides an extension mechanism to enable functions above what the Bayeux protocol supports. In some cases, an extension might be provided that is invalid. In this release, the server will ignore any unrecognized extensions.
400	The request could not be recognized by the server.	None.	The client sent a request that was not recognized by the server. There are a few reasons why this error message is displayed, including: <ul style="list-style-type: none"> <li>▶ The client sent a request that contains invalid JSON.</li> <li>▶ The JSON request was not send as the message field in a form encoding post request.</li> </ul>
401	The client ID specified is unknown.	The client ID that is specified.	This message is given when a server does not recognized the client ID. In most cases, this error message will be given with the Bayeux advice mechanism, telling the client to rehandshake and resubscribe to any previous topics subscribed to. This error message can be thrown for a few reasons: <ul style="list-style-type: none"> <li>▶ The client is blindly trying to connect to an invalid client ID.</li> <li>▶ There was server failure or session affinity failed in a clustered environment.</li> <li>▶ The client tried to connect past a Bayeux session timeout.</li> </ul>
402	The request is missing a required parameter.	The missing parameter.	The Bayeux protocol mandates required fields for many Bayeux messages. If a required field is missing, an error message will be thrown indicating which field is missing.
403	The channel specified is forbidden.	The channel that is forbidden.	If a client tried to subscribe or publish to a channel without the appropriate authorization permissions, this error will be sent back to the client.

Error code	Short description	Args	Description
404	The channel specified is unknown.	The unknown channel.	The client has tried to subscribe, unsubscribe, or publish to a channel that is unknown. This error is thrown most often when the multipleDestination channel format is specified and the first argument does not match an existing service integration bus topic space or topic space alias.
405	The channel syntax specified is invalid.	The invalid channel format.	This message is given when an invalid channel format is specified. Refer to the Bayeux protocol for information about valid Bayeux channel formats.
406	The extension field specified is invalid.	The invalid extension field.	This message is given when an invalid extension field is given. The Web messaging service implementation ignores supplied extension fields that it does not recognize, so this error message should not be seen by clients.
407	The publish request has failed.	The channel that publish request was issue on.	This error messages indicates that the publish request has failed. In most cases, this indicates that the clientCanPublish option has been set to false.
408	The client does not have authorization to connect to receive or publish messages.	None.	This error message is given during a handshake or connect request. In most cases, this error message is indicative of a problem configuring service integration bus security, in particular, granting the bus connector role for this incoming user.
500	The server encountered an internal error and could not complete the request.	None.	An internal server error has been encountered. Check the server logs for more information.

## 10.19 Sample: QuoteStreamer sample application

The Quote Streamer sample application uses the Web messaging service to simulate stock quotes to a Dojo-enabled client application. Simulated quotes are generated by a CommonJ timer that publishes messages to a service integration bus topic space. The Web messaging service links a Dojo-enabled client, a WebSphere Application Server internal message broker, and platform messaging for Web-based publication or subscription.

Communication is achieved through the Bayeux protocol. The Dojo Toolkit's cometd client links incoming JavaScript Object Notation messages to the Dojo event and topic system for processing. In the QuoteStreamer sample application, multiple Dojo widgets are specified in a market report summary article in HTML format.

### **Prerequisites and limitations**

Ensure that the required installation steps have been taken, as specified in Chapter 4, “Installation” on page 43. Take particular note of the iFix for messaging.

You must have a valid license for the application server listed (or use a valid trial version) and at least one of the browser clients listed to make use of the application. Other browsers, such as Opera or Safari might also work, but they have not been extensively tested with the application.

The application servers supported are:

- ▶ WebSphere Application Server Version 6.0.x or Version 6.1.x
- ▶ WebSphere Application Server - Express Version 6.0.X or 6.1.X

The browser clients supported are:

- ▶ Microsoft Internet Explorer Version 6.0
- ▶ Microsoft Internet Explorer Version 7.0
- ▶ Firefox 2.0.X
- ▶ Mozilla 1.7.X

The QuoteStreamer sample as provided does not include the capabilities to run in a secure environment. The QuoteStreamer application can be secured by following instructions in 10.15, “Securing Web messaging enabled applications” on page 496. The sample should not be installed in a clustered environment. If you do install the QuoteStreamer sample, each cluster member publishes messages to Web messaging clients. The application will still function, but you will see a higher message rate proportional to the number of cluster members.

## **10.19.1 Pre-installation configuration**

These configuration instructions assume WebSphere Application Server Version 6.1.x. If instructions differ significantly on a different application server version, we note it. Instructions marked by an asterisk (\*) denote steps that are applicable to version 6.1.x only.

## Enable the Web messaging service

To enable the Web messaging service:

1. Log into the administration console of WebSphere Application Server.
2. Navigate to **Servers** → **Application Servers**, as shown in Figure 10-10.

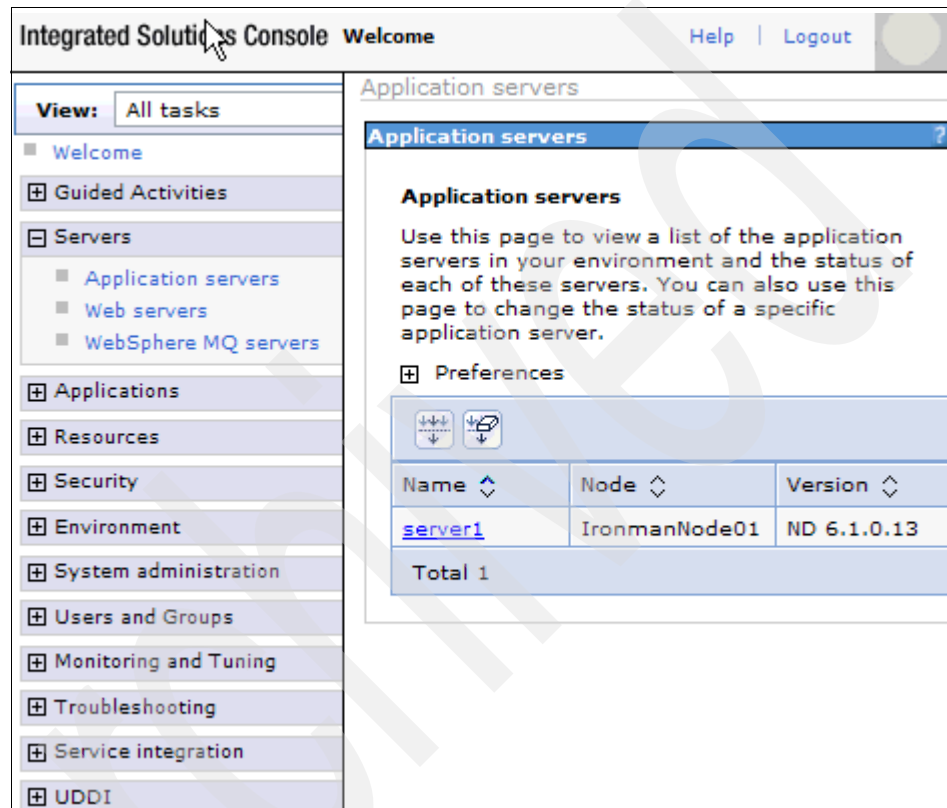


Figure 10-10 Navigate to Servers → Application Servers

3. Select the current application server.

- Expand **Web Container Settings** and select **Web container transport chains**, as shown in Figure 10-11.

**Application servers > server1**

Use this page to configure an application server. An application server is a server that provides services required to run enterprise applications.

**Runtime** **Configuration**

**General Properties**

Name: server1

Node Name: IronmanNode01

☐ Run in development mode

☒ Parallel start

**Container Settings**

- Session management
- SIP Container Settings
- Web Container Settings**
  - Web container
  - Web container transport chains**

Figure 10-11 Select Web container transport chains

- Select the **WCInboundDefault** transport chain or select the transport chain that corresponds to the virtual host that you will select during application install. See Figure 10-12.

<div>New Delete</div> <div> <input checked="" type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> <input type="checkbox"/> </div>					
Select	Name	Enabled	Host	Port	SSL Enabled
<input type="checkbox"/>	<a href="#">WCInboundAdmin</a>	Enabled	*	9060	Disabled
<input type="checkbox"/>	<a href="#">WCInboundAdminSecure</a>	Enabled	*	9043	Enabled
<input type="checkbox"/>	<a href="#">WCInboundDefault</a>	Enabled	*	9080	Disabled
<input type="checkbox"/>	<a href="#">WCInboundDefaultSecure</a>	Enabled	*	9443	Enabled
Total 4					

Figure 10-12 Select WCInboundDefault (if appropriate)

6. Select **Web Container Inbound Channel**, as shown in Figure 10-13.

**General Properties**

\* Name  
WCInboundDefault

☒ Enabled

**Transport Channels**

- ◆ [TCP inbound channel \(TCP\\_2\)](#)

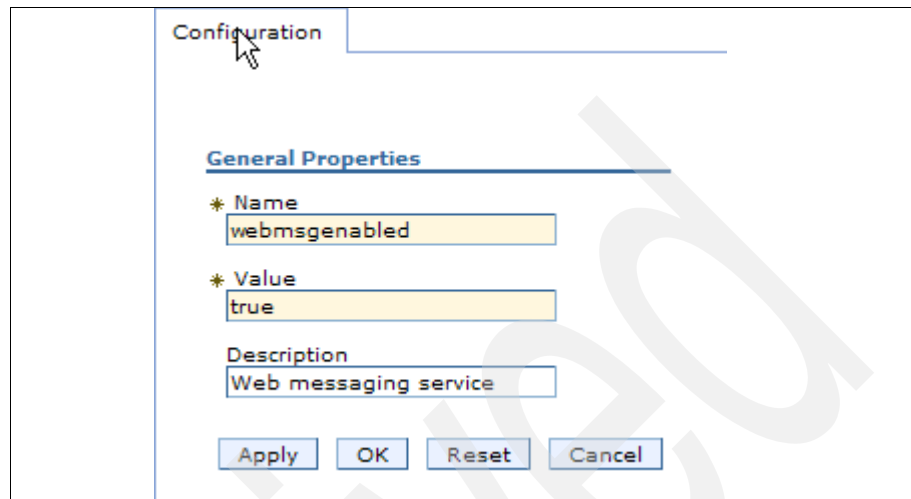
Host	*
Port	9080
Thread pool	WebContainer
Maximum open connections	20000
Inactivity timeout	60 seconds
- ◆ [HTTP inbound channel \(HTTP\\_2\)](#)

Use persistent (keep-alive) connections	Enabled
Maximum persistent requests per connection	100
Read timeout	60 seconds
Write timeout	60 seconds
Persistent timeout	30 seconds
- ◆ [Web container inbound channel \(WCC\\_2\)](#)

Figure 10-13 Select Web container inbound channel

7. Select **Custom Properties**.
8. Click **New**.

9. Enter `Webmsgenabled` for the name property and `true` for the value, as shown in Figure 10-14.



The screenshot shows a 'Configuration' dialog box with a 'General Properties' section. The 'Name' field contains 'webmsgenabled' and the 'Value' field contains 'true'. The 'Description' field contains 'Web messaging service'. At the bottom are buttons for 'Apply', 'OK', 'Reset', and 'Cancel'. A mouse cursor is pointing at the 'Configuration' tab.

General Properties	
* Name	webmsgenabled
* Value	true
Description	Web messaging service

Buttons: Apply, OK, Reset, Cancel

Figure 10-14 Creating the `Webmsgenabled` custom property

10. Click **Apply**.
11. Click **Save** to save the repository information.
12. You can restart the application server now or wait until after the next step ("Configure a service integration bus" step below).

### Configure a service integration bus

These instructions assume a standalone server installation. Instructions may differ for a Network Deployment environment:

1. Log into the administrative console of WebSphere Application Server.



2. Navigate to **Service integration** → **Buses** in the left menu, as shown in Figure 10-15.



Figure 10-15 Select Service integration → Buses

3. Click **New**.

4. Enter the name thebus. Ensure that the Bus security check box option is *not* checked. See Figure 10-16. Click **Next**.

Create a new messaging engine bus

Create a new Service Integration Bus.

→ Step 1: Create a new bus

Step 2: Confirm create of new bus

Create a new bus

Configure the attributes of your new bus

\* Enter the name for your new bus.

thebus

Bus security

☐

Next Cancel

Figure 10-16 Creating a new messaging engine bus

5. Click **Finish** on the Confirm create bus panel, as shown in Figure 10-17.

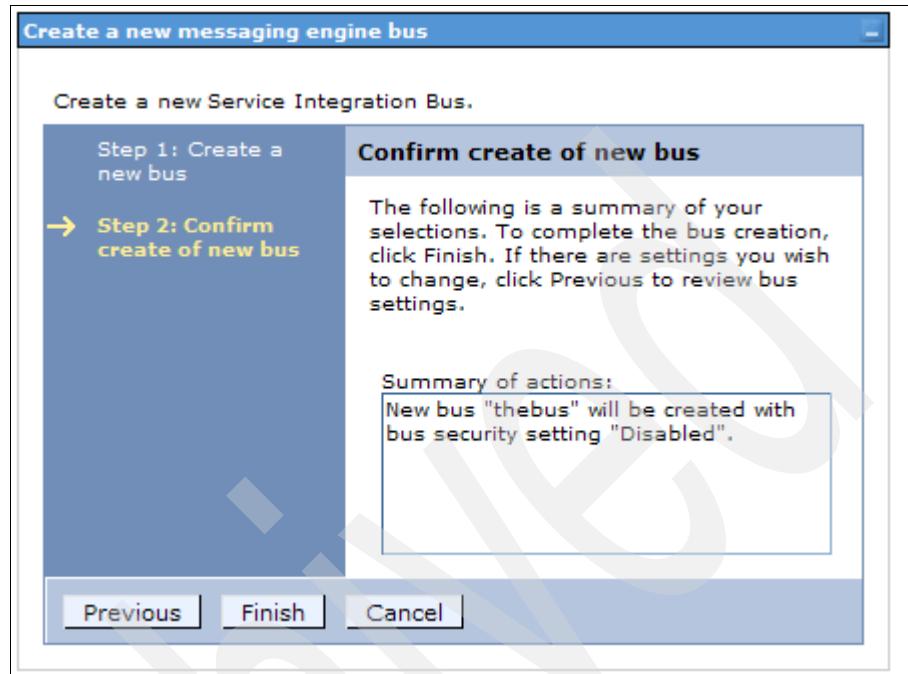


Figure 10-17 Confirming the new bus creation

6. Click **thebus** to access the bus detail page, as shown in Figure 10-18.

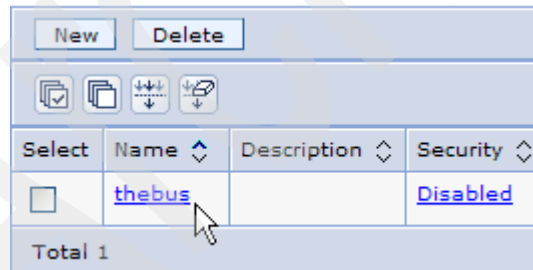


Figure 10-18 Select thebus link

- Click **Bus members** on the thebus detail page, as shown in Figure 10-19.

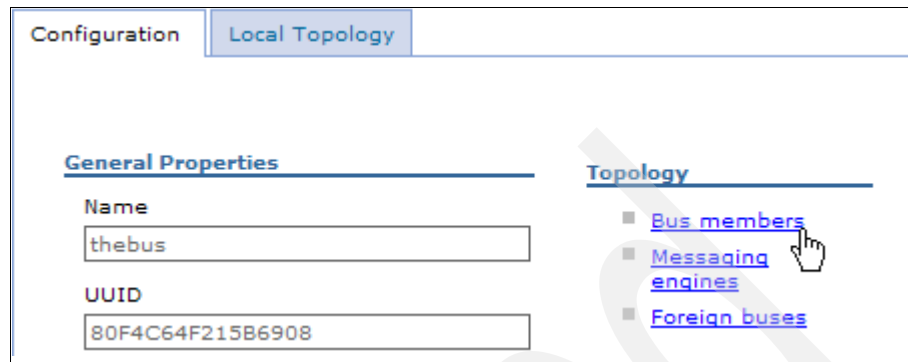


Figure 10-19 Click bus members

- Click **Add**.
- Select the server where you want the application installed and click **Next**, as shown in Figure 10-20.

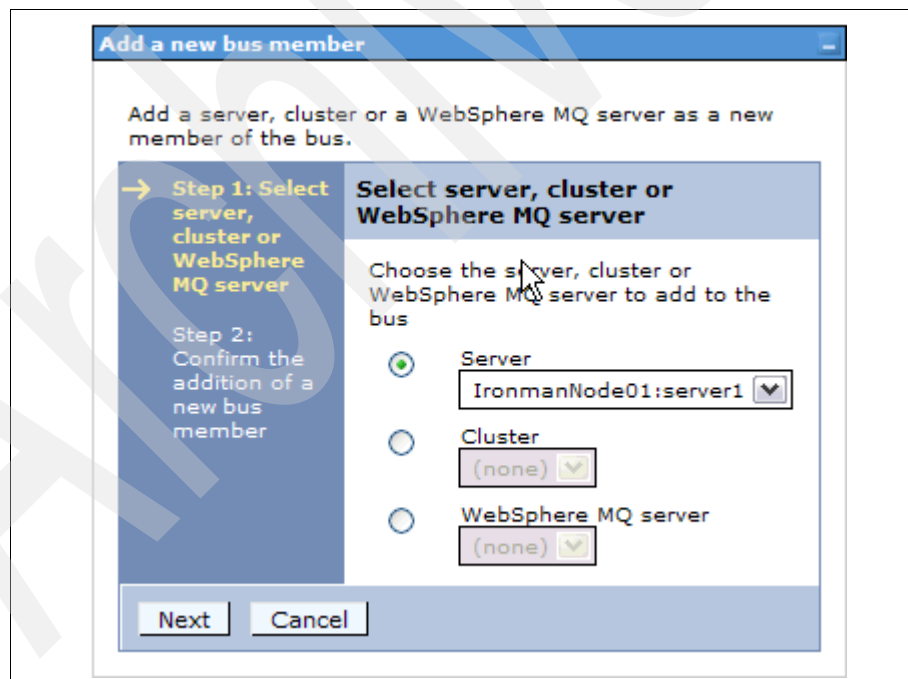


Figure 10-20 Select the server

- Keep the default on the Select type of Message Store panel and click **Next**.\*

11. Keep the default on the Provide the Message Store properties panel and click **Next**.\*
12. Click **Finish**, as shown in Figure 10-21.

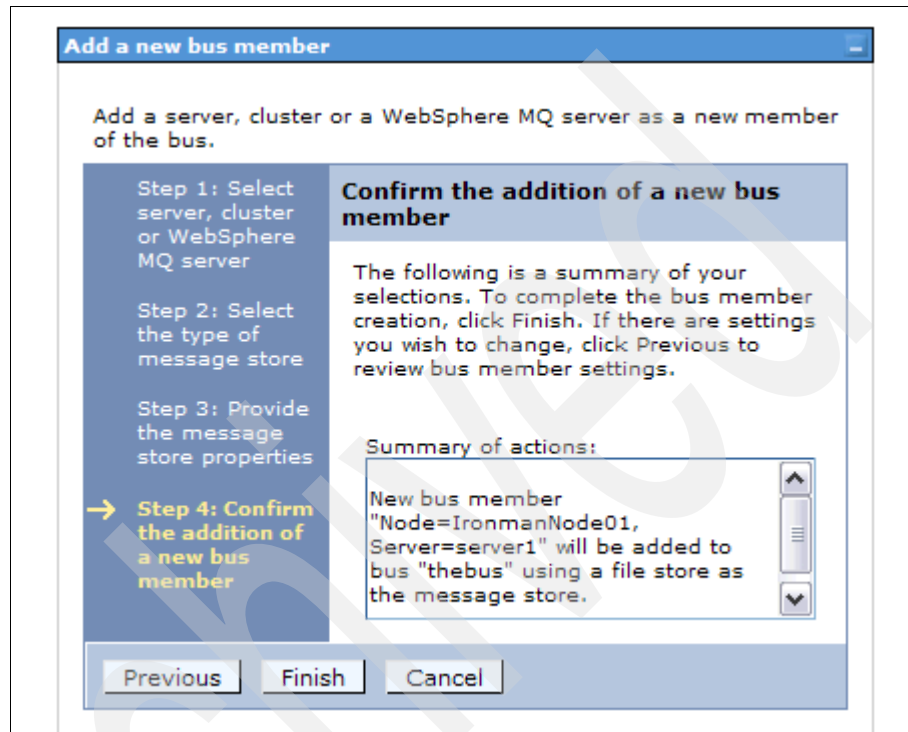


Figure 10-21 Adding a new bus member

13. Save the changes to repository.
14. Restart the application server.

## Create a topic connection factory

To create a topic connection factory:

1. Log into the administrative console of WebSphere Application Server.

2. Navigate to **Resources** → **JMS** → **JMS Providers** → **Default messaging**, as shown in Figure 10-22.

**Integrated Solutions Console** Welcome [Help](#) | [Logout](#)

**View:** All tasks

- Welcome
- ⊕ Guided Activities
- ⊕ Servers
- ⊕ Applications
- ⊖ Resources
  - Schedulers
  - Object pool managers
  - ⊖ JMS
    - JMS providers
    - Connection factories
    - Queue connection factories
    - Topic connection factories
    - Queues
    - Topics
    - Activation specifications
  - ⊕ JDBC
  - ⊕ Resource Adapters
  - ⊕ Asynchronous beans
  - ⊕ Cache instances

### JMS providers

**JMS providers**

A JMS provider enables messaging based on the Java Message Service (JMS). It provides J2EE connection factories to create destinations.

⊖ Scope: =All scopes

Scope specifies the level at which the resource is visible. For detailed information on what scope works, [see the scope settings help](#).

All scopes

⊕ Preferences

New Delete

Select	Name	Description	Scope
<input type="checkbox"/>	<a href="#">Default messaging provider</a>	Default messaging provider	Cell=Ironm

Figure 10-22 Select Resources → JMS → JMS providers → Default messaging provider

3. Select the **Topic connection factories** link under Additional Properties, as shown in Figure 10-23.

The screenshot shows a 'Configuration' dialog box with two main sections: 'General Properties' and 'Additional Properties'. The 'General Properties' section contains three text input fields: 'Scope' with the value 'Cell=IronmanNode01Cell', 'Name' with the value 'Default messaging provider', and 'Description' with the value 'Default messaging provider'. The 'Additional Properties' section is on the right and contains a list of links: 'Connection factories', 'Queue connection factories', 'Topic connection factories', 'Queues', 'Topics', and 'Activation specifications'. A mouse cursor is pointing at the 'Topic connection factories' link. An 'OK' button is located at the bottom left of the dialog box.

Figure 10-23 Select Topic connection factories

4. Click **New**.

5. Enter QuoteStreamer for name, jms/QuotePublish for JNDI name, and thebus for bus name, as shown in Figure 10-24.

The screenshot shows a 'Configuration' window with a 'General Properties' tab. Under the 'Administration' section, the following fields are visible: 'Scope' with the value 'Cell=IronmanNode01Cell', 'Provider' with the value 'Default messaging provider', '\* Name' with the value 'QuoteStreamer', '\* JNDI name' with the value 'jms/QuotePublish', an empty 'Description' text area, and an empty 'Category' text field. Under the 'Connection' section, the '\* Bus name' dropdown menu is set to 'thebus'.

Figure 10-24 Creating a new topic connection factory

6. Keep the defaults for the rest of the fields. Click **OK**.
7. Click **Save** to save the repository information.



## 10.19.2 Installation through the administrative console

To install:

1. Log into the administrative console of WebSphere Application Server.
2. Navigate to **Application** → **Install New Application**, as shown in Figure 10-25.

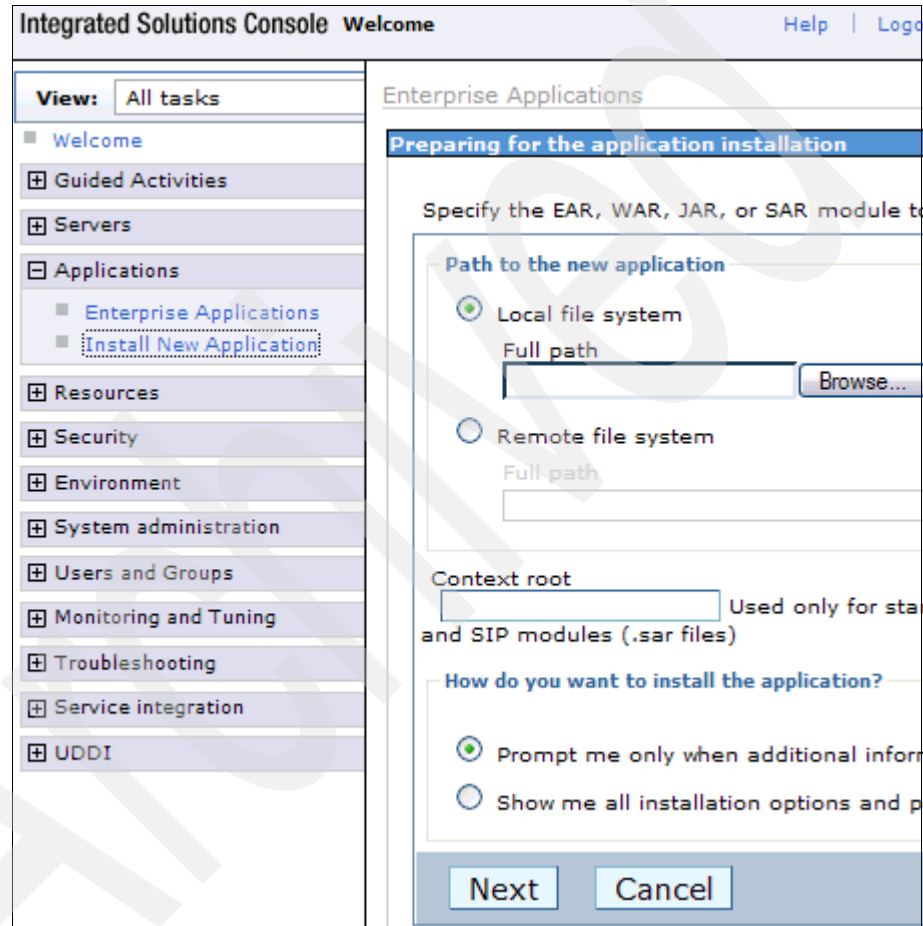


Figure 10-25 Installing QuoteStreamer through the administrative console

3. Browse your file system and select the **IBM\_QuoteStreamer.ear** file.

The IBM WebSphere Application Server Feature Pack for Web 2.0 installer places the QuoteStreamer EAR into the folder located at `<app_server_root>/Web2fp/samples/QuoteStreamer`. Click **Next**.

The stand-alone package includes the QuoteStreamer WAR file inside the folder located at

`<app_server_root>/AjaxClientRuntime_1.X/QuoteStreamer/samples.`

4. On the Select installation options panel keep the defaults and click **Next**.
5. On the Map modules to server panel, click **Next**.
6. Click **Finish**.
7. Click **Save directly to the master configuration**. See Figure 10-26.

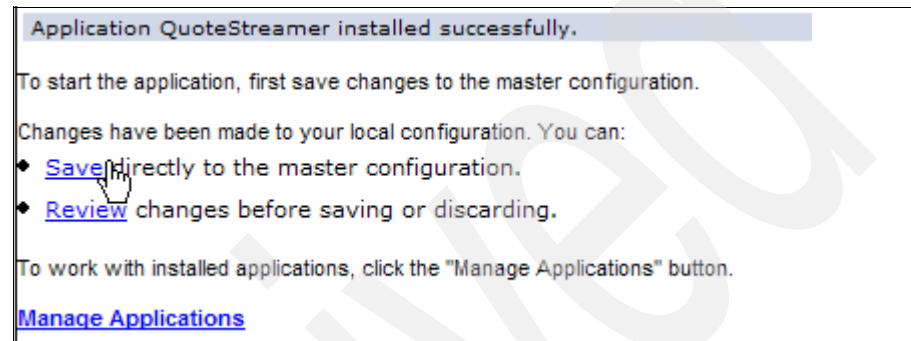


Figure 10-26 Installing the QuoteStreamer EAR

8. Select the QuoteStreamer project and click **Start**, as shown in Figure 10-27.

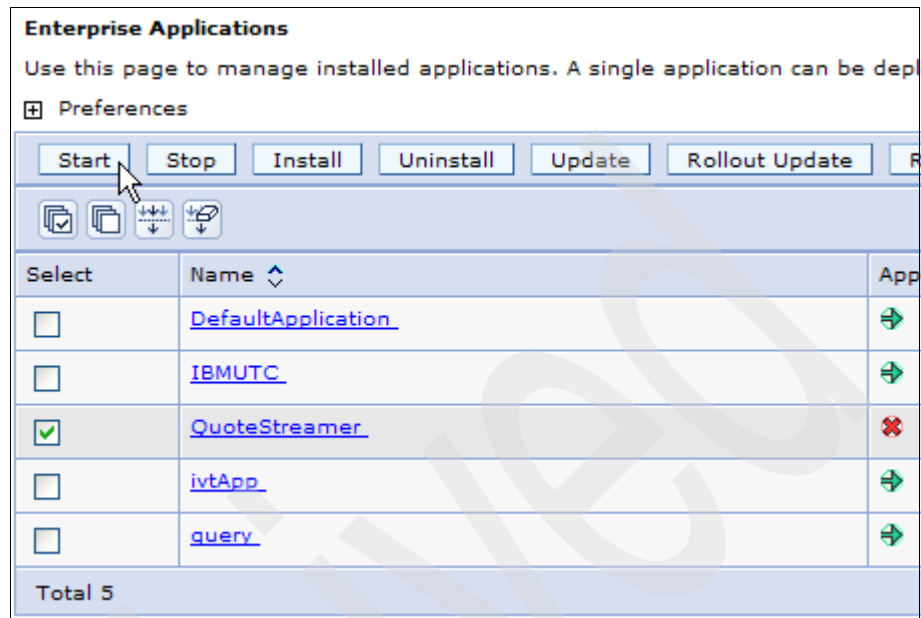


Figure 10-27 Starting the QuoteStreamer sample application

You should see a successful start message and the started icon beside the QuoteStreamer application.

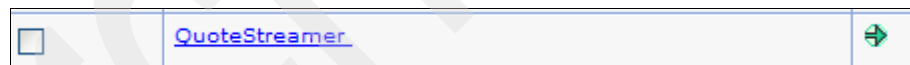


Figure 10-28 QuoteStreamer sample application successfully started

You can see the sample in action by following the instructions in 10.19.4, “Accessing to the installed application” on page 523.

### 10.19.3 Installation through Rational Application Developer

These instructions assume that the Eclipse plug-in for Feature Pack for Web 2.0 has been installed (see 5.2.5, “Configuring Web 2.0 feature pack Eclipse plug-in” on page 78).

1. Open Rational Application Developer in a new workspace.
2. Click **File** → **New** → **Other** from the workbench toolbar.
3. Expand **Examples** → **IBM WebSphere Application Server Feature Pack for Web 2.0 samples** in pop-up window.

4. Select **QuoteStreamer Sample Application for WebSphere Application Server** and click **Next**, as shown in Figure 10-29.

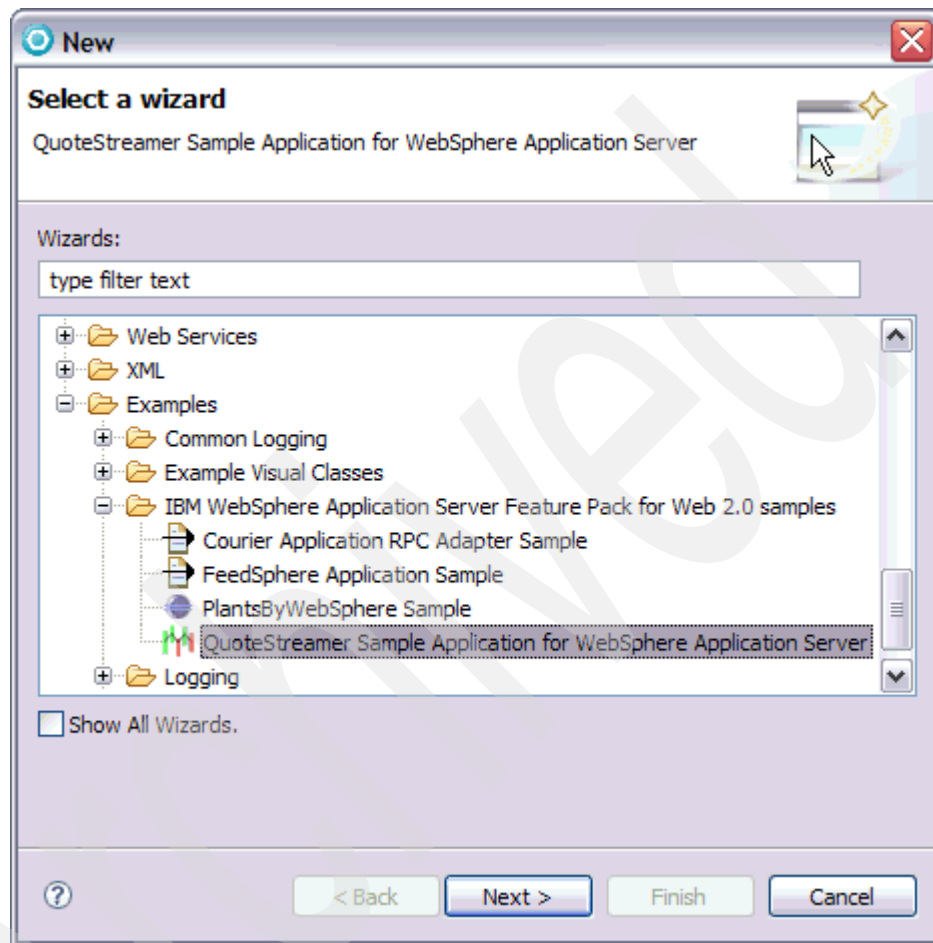


Figure 10-29 Select the QuoteStreamer sample application and click Next

5. Select **WebSphere Application Server v6.1** as the target run time from the drop-down list. See Figure 10-30.  
Create the server run time if it does not already exist.

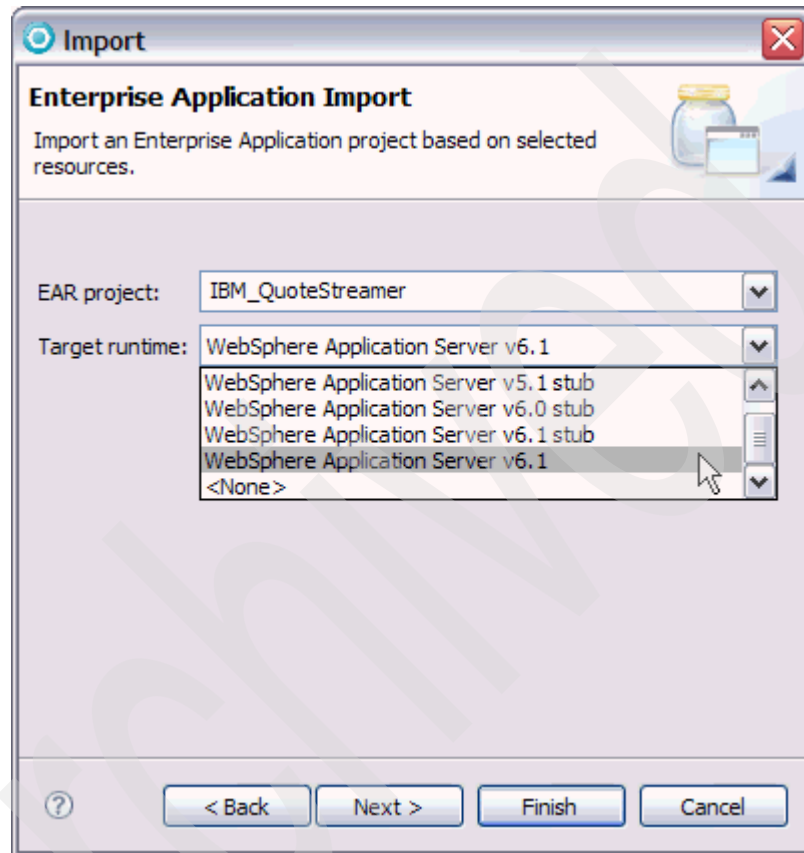


Figure 10-30 Selecting the target run time

Click **Next**.

6. Click **Finish**.

The IBM\_QuoteStreamer project is now created. See Figure 10-31.

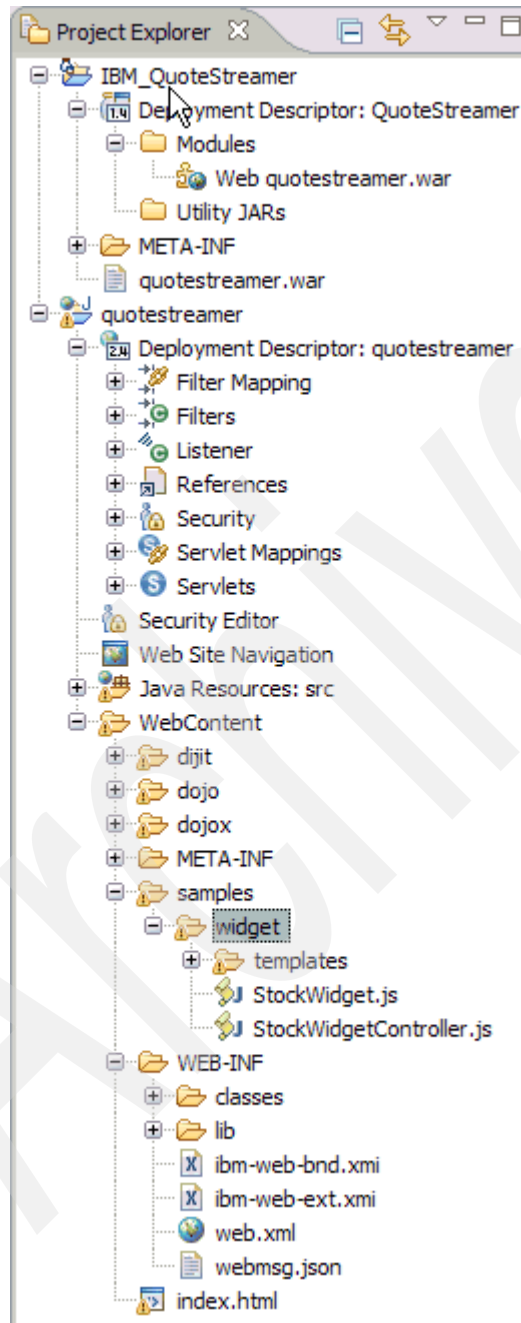


Figure 10-31 Project Explorer view of QuoteStreamer sample application

The QuoteStreamer sample should now publish to the server.

### 10.19.4 Accessing to the installed application

Open your browser to the following address:

`http://<app_server_hostname>:<port>/quotestreamer`

**Note:** The application server host name and port will be specific to your WebSphere Application Server installation. A WebSphere Application Server default install Web container port is 9080:

`http://localhost:9080/quotestreamer`

In the QuoteStreamer sample application, multiple Dojo widgets are specified in a market report summary article in HTML format. These Dojo widgets process incoming stock quote messages and visually indicate stock changes. These visual changes include:

- Updates to the current price of a stock
- Daily stock price change
- Daily stock percent price change

The QuoteStreamer sample application output is shown in Figure 10-32.

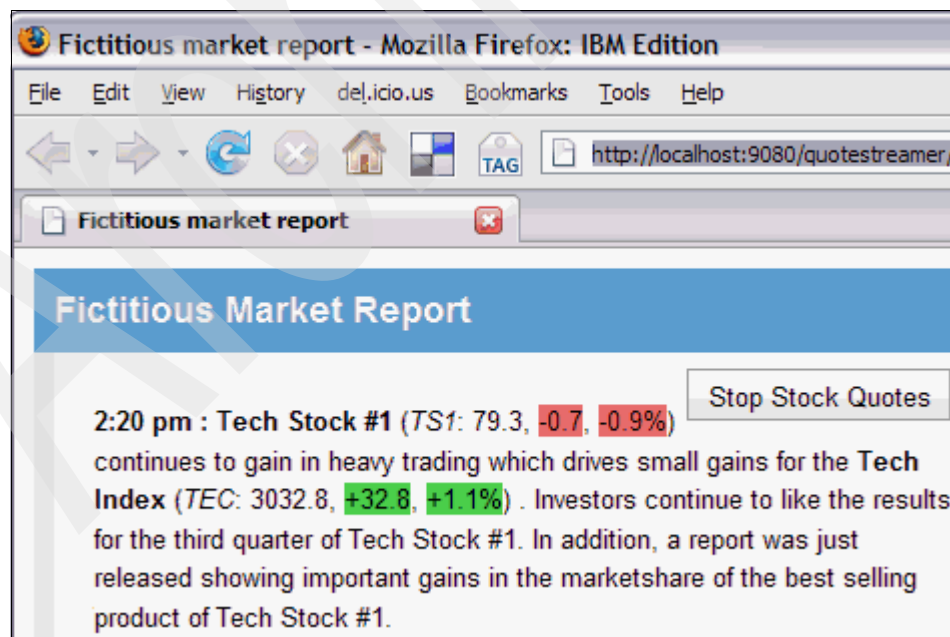


Figure 10-32 Simulated quotes being delivered to multiple custom Dojo widgets

When the price of a stock changes, the daily price change and daily percent price change represents green (increase) or red (decrease) before returning to the original background color, as shown in Figure 10-33. Simulated quotes will be delivered to the browser immediately upon page load. Clicking the **Stop stock quotes** button stops incoming quotes. Clicking the **Resume stock quotes** button resumes the quotes.

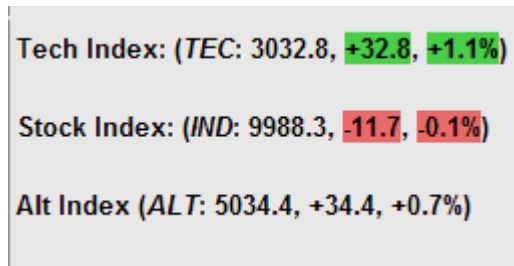


Figure 10-33 Dojo widgets being updated by the Web messaging service

### 10.19.5 How the QuoteStreamer sample application works

In this section we discuss how the QuoteStreamer sample application works.



## Server-side

The QuoteStreamer sample application uses the Webmsg\_applib.jar Web messaging service library. The contents are shown in Figure 10-34.

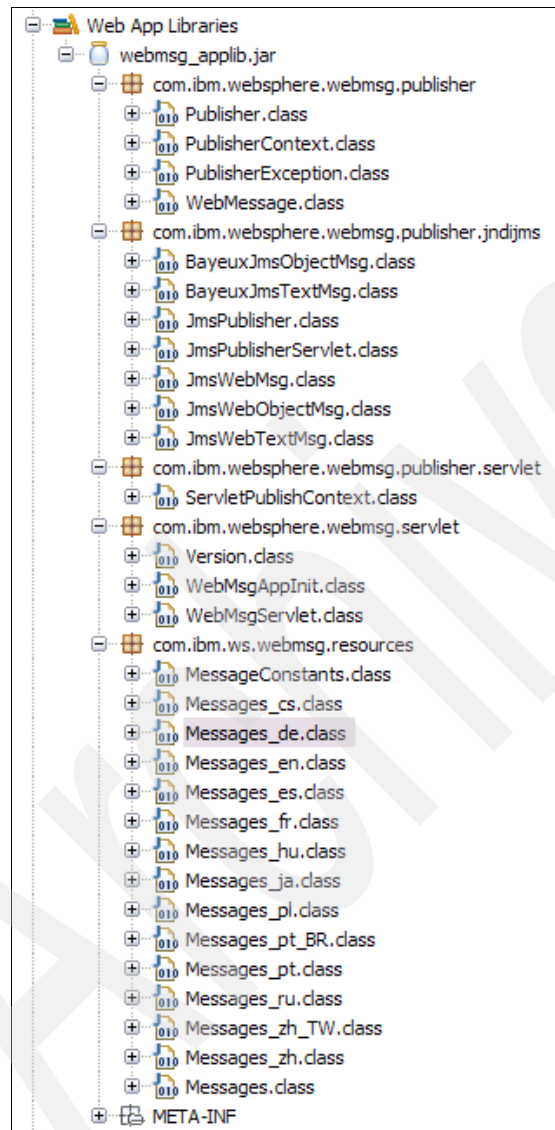


Figure 10-34 The Web messaging service library

The QuoteStreamer sample application's Web.xml file shown below details the servlets that are loaded on application startup:

```
<servlet>
  <description></description>
  <display-name>WebMsgAppInit</display-name>
  <servlet-name>WebMsgAppInit</servlet-name>
  <servlet-class>
    com.ibm.websphere.Webmsg.servlet.WebMsgAppInit
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet>
  <description></description>
  <display-name>Publisher</display-name>
  <servlet-name>Publisher</servlet-name>
  <servlet-class>
    com.ibm.websphere.Webmsg.publisher.jndijms.JmsPublisherServlet
  </servlet-class>
  <init-param>
    <description></description>
    <param-name>CONNECTION_FACTORY_JNDI_NAME</param-name>
    <param-value>java:comp/env/jms/QuotePublish</param-value>
  </init-param>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet>
  <description></description>
  <display-name>AppInit</display-name>
  <servlet-name>AppInit</servlet-name>
  <servlet-class>
    com.ibm.websphere.Webmsg.quotestreamer.AppInit
  </servlet-class>
  <load-on-startup>2</load-on-startup>
</servlet>
```

The load-on-startup field specifies that the WebMsgAppInit, Publisher, and AppInit servlets are loaded in that order.

The WebMsgAppInit servlet provides the basic initialization of the QuoteStreamer Web messaging service.

Publisher initializes the JMS Publishing service by providing the JNDI name of the QuoteStreamer Topic Connection factory (which is connected to our “thebus” service integration bus) that we set up through the administrative console earlier.

Finally, the sample application's Applnit servlet is initialized. The of the init() method of Applnit.java are:

```
super.init();
```

```
List simData = new ArrayList(10);
```

```
simData.add(
    new StockData("Tech Stock #2", "/stocks/TS2", 55, 25, 1, 49));
simData.add(
    new StockData("Tech Stock #1", "/stocks/TS1", 80, 25, 1, 51));
simData.add(
    new StockData("Drug Stock #1", "/stocks/DS1", 10, 38, 10, 49));
simData.add(
    new StockData("Drug Stock #2", "/stocks/DS2", 40, 35, 10, 49));
simData.add(
    new StockData("Stock Index", "/stocks/IND", 10000, 40, 1, 51));
simData.add(
    new StockData("Tech Index", "/stocks/TEC", 3000, 30, 1, 49));
simData.add(
    new StockData("Alternative Index", "/stocks/ALT", 5000, 35, 1, 50));

try {
    DataSimulatorAppHelper.setDataSimulator(
        getServletConfig().getServletContext(),
        simData,
        "java:comp/env/jms/QuotePublish",
        3,
        true);
    logger.log(Level.INFO, "DataSimulator successfully created and set
and started.");
} catch (RuntimeException e) {
    logger.log(
        Level.WARNING,
        "Unable to create and initialize DataSimulator. Reason: " +
        e.getMessage()
    );
}
```

The `init()` method adds `StockData` objects to a collection. `StockData` is a class that implements the `SimulatedData` interface and represents the relevant stock information. The `StockData` objects simulate realistic stock price movement and create the JSON formatted output to be published. The `init()` method instantiates seven `StockData` objects (four representing stocks, three representing stock indices). Each instantiation of a `StockData` object is passed:

- ▶ Name string
- ▶ Stock symbol string used for topics
- ▶ Ints for:
  - Starting price
  - Percentage chance-of-change number (for simulating volatility)
  - Percentage change-range number (for price movement)
  - Trend indicator (the general price direction)

Then the `DataSimulatorAppHelper`'s `setDataSimulator` static method will set up the data simulation by first retrieving the publisher object that has been set up previously when the `JmsPublisherServlet` was initialized. It then creates a new instance of `DataSimulatorRunner` passing in the publishing interval (in seconds) number (in this case, 3 seconds) and a new instance of `DataSimulator`, passing in the Publisher object and collection of `StockData` objects, as shown here:

```
/* Get DataPublisher to publish through SIBus */
Publisher publisher =
    (Publisher)servletContext.getAttribute(
        JmsPublisherServlet.PUBLISHER_SERVLET_CONTEXT_KEY);
...
DataSimulatorRunner runner =
    new DataSimulatorRunner(
        interval,
        new DataSimulator(publisher, simData)
    );

// Store reference to object in servletcontext in this version
servletContext.setAttribute(APPLICATION_KEY, runner);

if (startImmediately) {
    runner.startDataSimulator();
}
```

The `DataSimulator` class is responsible for publishing the prices when the timer expires every interval seconds through the service integration bus. The `DATA_SIMULATOR` servletContext attribute is then associated with the `DataSimulatorRunner` object and, as `startImmediately` is true when passed from `AppInit`'s `init()` method, it begins simulating real-time stock prices immediately.

QuoteStreamer uses the CommonJ library for timing the message publishing. The CommonJ TimerManager has been configured in the application's deployment descriptor. See Figure 10-35.

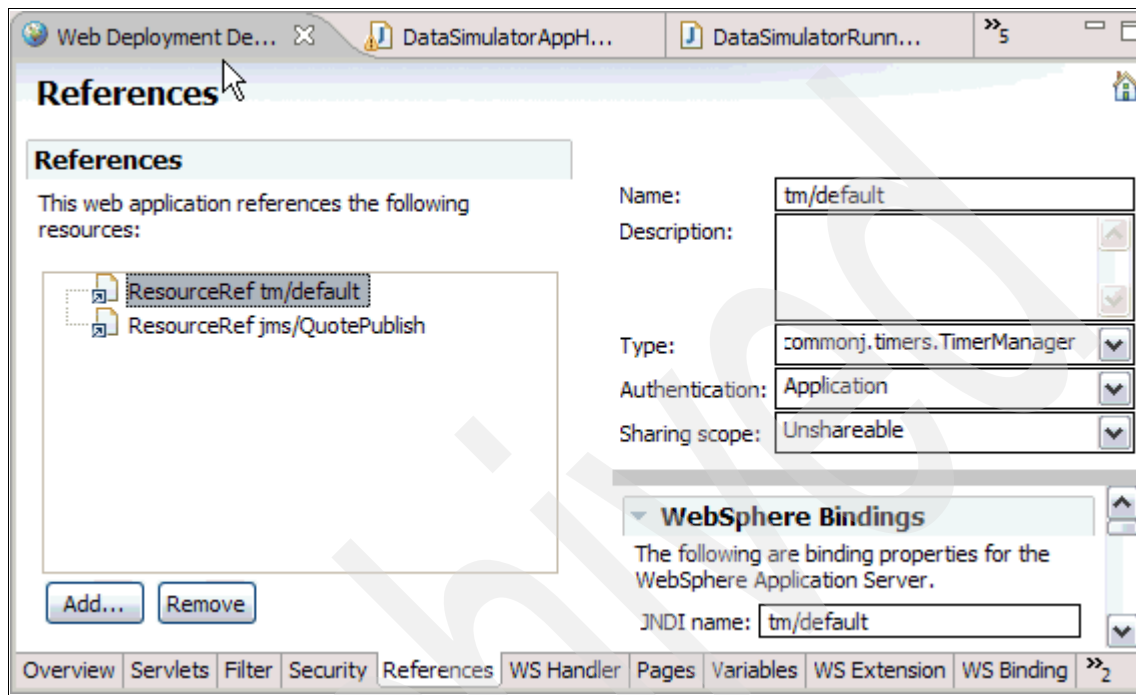


Figure 10-35 The CommonJ TimerManager resource configured in Web.xml

When the DataSimulatorRunner is started, the JNDI lookup returns the TimerManager reference, a ChangeListener is placed on the DataSimulator publisher object (simulator), and the TimerManager schedules the listener to execute every 3 seconds, after an initial delay of 5 milliseconds, as shown here:

```
public void startDataSimulator() {
    synchronized (this) {
        if (running == true) {
            return;
        } try {
            running = true;
            InitialContext ic = new InitialContext();
            tm =
                (TimerManager) ic.lookup("java:comp/env/tm/default");
            TimerListener listener = new ChangeListener(simulator);
            tm.scheduleAtFixedRate(
                listener,
```

```

        5,
        quoteInterval * 1000
    );
} catch (NamingException e) {
    throw new RuntimeException(
        "DataSimulator error starting timer. " +
        "Reason: " + e.toString(), e);
}
}
}

```

ChangeListener is an inner class of the DataSimulatorRunner class that extends the CommonJ TimerListener listener class. When the timer interval expires, the timerExpired method is called:

```

public void timerExpired(Timer timer) {
    simulator.timerExpired();
}

```

This in turn calls the DataSimulator object's timerExpired() method. This method iterates over the collection of StockData objects contained within the DataSimulator class' List object (data). For each StockData object it gets the returned JSON-formatted data (if any) as the object attempts to simulate the stock's price changing. If the criteria to simulate the stock's price changing has been fulfilled and data is returned, the DataSimulator object publishes a new BayeuxJmsTextMsg object, supplying the stock's name (topic) as the Bayeux channel to publish on and the returned JSON data as the associated data, as shown here:

```

public void timerExpired() {
    Iterator i = data.iterator();
    while (i.hasNext()) {
        SimulatedData simData = (SimulatedData) i.next();
        String data = simData.simulateChange();
        if ((data != null) && (!"".equals(data)) ) {
            try {
                publisher.publish(
                    new BayeuxJmsTextMsg(simData.getTopic(), data)
                );
            } catch (PublisherException e) {
                // Log and trace as you please
            }
        }
    }
}

```

How the StockData object's simulateChange() method figures out whether the stock's price has changed and by how much is not important here, but what is notable is that the simulateChange() method does call the getJSONChangeString() method to return JSON-formatted text to the data simulator, as shown here:

```
return "{\"currentPrice\":\" + roundCurPrice +  
    \", \"change\":\" + roundLastChange +  
    \", \"percentChange\":\" + roundLastPercentChange +  
    \", \"gaining\":\" + gaining +  
  
    }\";
```

The Web messaging library's BayeuxJmsTextMsg object takes this JSON data and puts it in the Bayeux event delivery message data field of the JMS text message object.

## Client-side

We use Firebug (see 5.4, “Firebug 1.05” on page 92) to view the traffic between client and server. Enable and open the Firebug console, then open your browser to the QuoteStreamer application, as in 10.19.4, “Accessing to the installed application” on page 523. The welcome page, index.html, is loaded. The first HTML script tag within index.html contains the following:

```
<script type="text/javascript">  
    function myInit() {  
        dojox.cometd.init("stockServlet");  
    }  
    ...  
    dojo.addOnLoad(myInit);  
</script>
```

On page loading, the Dojo cometd client initializes the stockServlet URI, which is mapped in Web.xml to the servlet class com.ibm.websphere.Webmsg.servlet.WebMsgServlet. This servlet is provided in the Web messaging utility library, as shown here:

```
<servlet>  
    <description></description>  
    <display-name>WebMsgServlet</display-name>  
    <servlet-name>WebMsgServlet</servlet-name>  
    <servlet-class>  
        com.ibm.websphere.Webmsg.servlet.WebMsgServlet  
    </servlet-class>  
</servlet>  
<servlet-mapping>  
    <servlet-name>WebMsgServlet</servlet-name>
```

```

        <url-pattern>/stockServlet</url-pattern>
    </servlet-mapping>

```

The Web messaging service reads configuration parameters from a configuration file to handle incoming Web messaging requests. It will attempt to locate the Webmsg.json file in the same directory as the Web.xml descriptor, match an entry to the servlet name (WebMsgServlet), and load the configuration options into the servlet, as shown here:

```

{
    "WebMsgServlet":
    {
        "busName": "thebus",
        "destination": "Default.Topic.Space",
        "clientCanPublish": false,
        "longPollTimeout": 30
    }
}

```

The Dojo cometd client initialization starts the Bayeux messaging between the client and the server. The initial step is to set up the communication connection with the Bayeux handshake by POSTing to the URI mapped to the Web messaging service servlet and supplying JSON data to negotiate the connection within a message envelope. See Figure 10-36.

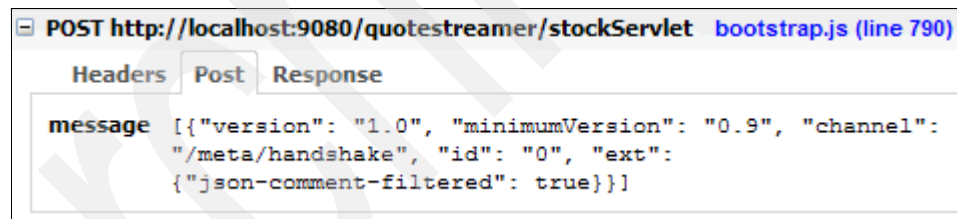


Figure 10-36 Setting up the Bayeux connection

Since we are using the meta channel, we are communicating with the event router itself. The handshake response received is shown in Figure 10-37.

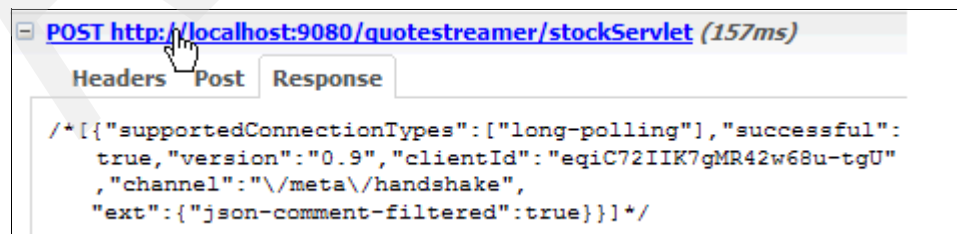


Figure 10-37 The response of the handshake



The handshake response informs the client of the server capabilities and assigns a client ID to this connecting client. The client then attempts to establish a long-polling request with the server, as shown in Figure 10-38.

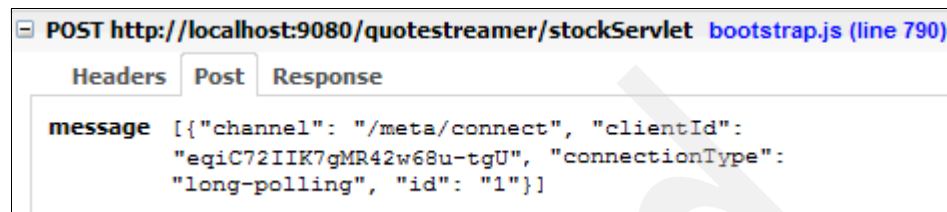


Figure 10-38 Establishing the connection method

The connect response is shown in Figure 10-39.

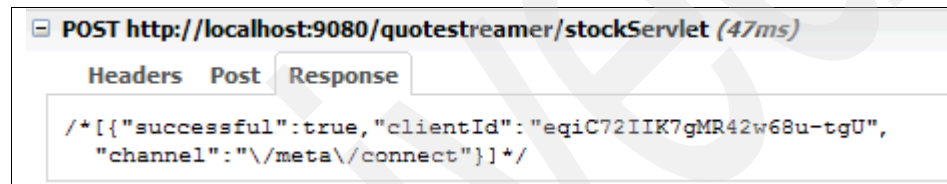


Figure 10-39 The connect response

The response indicates that the connection has been established on the `/meta/connect` channel.

## The StockWidget custom Dojo widget

The console initially shows the Dojo Toolkit client runtime files being fetched, including the custom widget (`StockWidget.js`), its associated controller (`StockWidgetController.js`), and its HTML template (`StockWidget.html`).

The `StockWidget` custom Dojo widget is created declaratively within `index.html`:

```
<span dojoType="samples.widget.StockWidget"
  ticker="TS2"
  controlTopic="/stockWidgetControl"
  controller="swc"
  stockTopic="/stocks/TS2">
</span>
```

The widget's ticker represents the stock ticker, the controlTopic is used to start and stop the real-time price updates using the controller swc, and the stockTopic is the channel name on which this stock will receive its price updates. The controller (swc) is declared in index.html also:

```
<div id="swc"
      dojoType="samples.widget.StockWidgetController">
</div>
```

The StockWidget widget and template together display the stock name, current price, daily price change, and daily percentage price change, as shown in Figure 10-40.

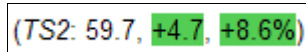


Figure 10-40 The StockWidget custom Dojo widget

When the price of a stock changes, the daily price change and daily percent price change represents green (increase) or red (decrease) before returning to the original background color. The widget accomplishes this as part of the widget's quoteUpdate function by connecting the dojo.fx.combine graphics effect function (which has specified the fields to be highlighted and progressively lightened with the highlight function) with the highlightComplete function. When the highlight function has ended and the fields' style's backgroundImage has returned to none, the highlighting has ended. For more information about the Dojo Toolkit, see Chapter 9, "Ajax client runtime" on page 269.

The widget's postCreate function registers the widget's associated stock topic (the Bayeux channel value) with the widget's quoteUpdate function, as well as registering the control topic (linked to the Start/Stop Stock Quotes button) with the control function (that handles the starting and stopping).

```
postCreate: function(){
    if(this.controlTopic) {
        this.controlSubscription = dojo.subscribe(
            this.controlTopic,
            this,
            "control"
        );
    }
    this.controller = digit.byId("swc");
    this.subscription = this.controller.register(
        this.stockTopic,
        this,
        "quoteUpdate"
    );
};
```

```

        this.hlightInProgress = false;
    },

```

It does this through the controller (StockWidgetController.js). The controller is responsible for managing the Bayeux subscription and unsubscription of the client to the server. When the widget's postCreate function is called, after the widget has been rendered on the window, the controller uses Dojo cometd to subscribe to the topic channel and to call the quoteUpdate function when data from the server arrives on that channel, as shown here:

```

subscribed: [],
register: function(topic, scope, fn){
    if(this.subscribed[topic]) {
        return dojo.subscribe("/cometd"+topic, scope, fn);
    } else {
        this.subscribed[topic] = true;
        dojox.cometd.subscribe(topic, scope, fn);
        return null;
    }
},

```

As all the widgets on index.html are being rendered, the cometd client will group the subscriptions together and POST them to the server, as shown in Figure 10-41.

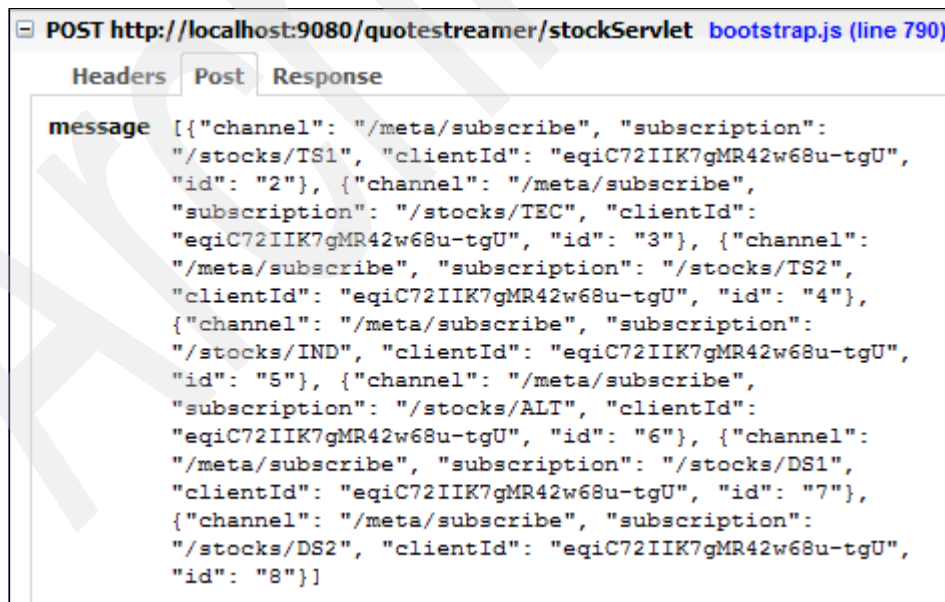


Figure 10-41 The widgets' subscription requests to the channels

The server responds with a field that indicates whether each widget subscribed to its desired channel successfully. See Figure 10-42.



Figure 10-42 The subscription response

The client then begins long polling the server (using the /meta/connect channel) until either the longPollTimeout configuration option's value (specified in Webmsg.json; in this case 30 seconds) has been reached or the server has pushed data to the client within this time frame. See Figure 10-43.

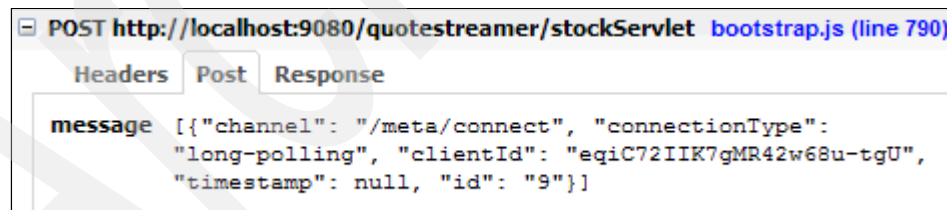


Figure 10-43 Long-polling the server

If it is the former, and the time out has expired without data being pushed from the server, the response is as shown in Figure 10-44, and a new long-polling request is established.

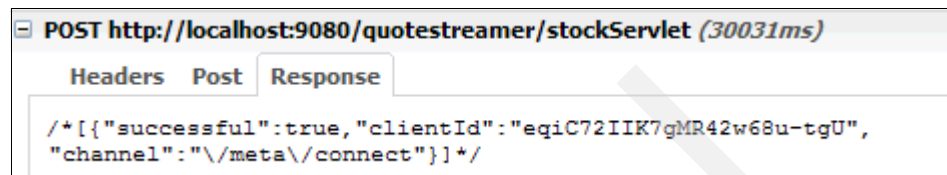


Figure 10-44 Response when no data pushed from server within 30 seconds

If it is the latter, the widget's quoteUpdate function is called, updating the widget's price data and applying the highlighting graphical effect. The data contained in the response (in the data field) has been populated by the getJSONChangeString() method, called from the StockData object's simulateChange() method, as described previously. See Figure 10-45.

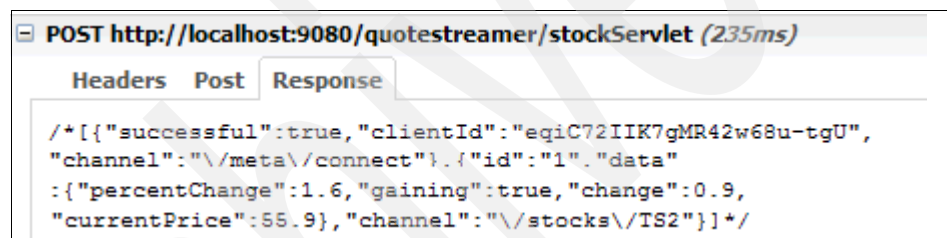


Figure 10-45 Server response of Bayeux message envelope containing price change data for TS2 stock



# Security

This chapter discusses common Web application security issues and their effects in context of Ajax. In this chapter we also introduce some of the tools available for securing Web 2.0 applications.

This chapter contains the following section:

- ▶ Attack Scenarios and the effect
- ▶ Recommended Best Practices
- ▶ Securing RPC Adapter (Application Server security)
- ▶ Tools: Datapower, Rational AppScan and Rational Application Developer

## 11.1 Overview

As Ajax is a group of inter-related Web development techniques used for creating an interactive Web application, it has changed the Web application development approach and methodology significantly and allows user interaction with Web pages to be decoupled from the Web browser's communication with the server. In particular, Ajax drives mashups, which integrate multiple contents or services into a single user experience, and so Ajax and mashup technology introduce new types of threats because of their dynamic and multidomain nature.

A Web 2.0 application can load several JavaScripts, Flash components, and widgets in the browser. These scripts and components utilize the XMLHttpRequest object to communicate with the backend Web server. It is also possible to access cross-domain information from within the browser itself.

## 11.2 Attack scenarios and the effect

In this section we discuss attack scenarios and their effects.

### 11.2.1 Cross-site scripting (XSS)

Cross-site scripting is one of the most common application-level attacks that hackers use to sneak into Web applications, as well as one of the most dangerous. Cross-site scripting refers to a large classification of vulnerabilities that focus on the ability to present malicious code to an unsuspecting user. Cross-site scripting occurs when an attacker introduces malicious scripts to a dynamic form that allows the attacker to capture private session information. An XSS attack leads to undesirable effects. For example, the attacker gains the ability to capture the session information and peer into private user details such as ID, passwords, credit card information, home address and telephone number, social security/tax IDs, and so on.

This usually takes the form of malicious JavaScript, which could do anything from annoying the user to capturing data from cookies. If the targeted Web site does not check for this type of malicious code, misuse of the user is probable.

To reduce the risk of having the script identified as malicious, the attacker might encode it with a different encoding method. With this alteration, the Web site displays the malicious content on the page as though the displayed information is the valid content from the site. If the Web application does not validate the input, all the attacker has to do is to coax the user to select the malicious hyperlink, after which the Web application collects confidential data from the user. This



enables the attacker to capture the user's session and steal the user's credentials, redirect to a page on another Web site, and then insert code that can poison cookies, expose SSL connections, access restricted or private sites, or even trigger a number of such attacks. The two basic types of XSS attacks are:

- ▶ Reflected XSS

A reflected XSS attack exploits vulnerable Web applications that display input parameters back to the browser without checking for the presence of active content in them.

- ▶ Stored XSS

The stored XSS attack has become more important with the prevalence of Web 2.0. The key of Web 2.0 is sharing, interaction, and collaboration among people, so users have more chance of seeing other (potentially malicious) users' input through services such as social network services (SNS), wikis, or blogs.

In either case, input value validation and sanitation are the key to preventing XSS attacks. Usually, Web servers remove scripts from user input, but often attackers exploit vulnerabilities to bypass these filters, resulting in major attacks.

XSS attacks are potential threats to the application user. The Web 2.0 framework uses various client-side scripts and may consume information from untrusted third-party sources. Ajax and JSON technologies, cross-domain access, and dynamic DOM manipulation techniques are adding new dimensions to XSS attacks such as JSON Injection, RSS and Atom Injection, DOM Injection along with SQL Injection, XML Injection, or HTML/JavaScript Injection. Client-side component scanning and vulnerability detection in Web 2.0 are new upcoming challenges.

## 11.2.2 Cross-site Request Forgery (CSRF)

Cross-Site® Request Forgery is an attack that allows a malicious user to perform an action on the vulnerable site on behalf of the victim. The attack is possible when a vulnerable site does not properly validate the origin of the request. The severity of this vulnerability depends on the functionality of the affected application. For example, a CSRF attack on a search page is less severe than a CSRF attack on a money-transfer or profile-update page. The attack is performed by forcing the victim's browser to issue an HTTP request to the vulnerable site. If the user is currently logged-in to the vulnerable site, the request will automatically use the user's credentials, such as session cookies. Using this method the attacker forges the victim's identity and submits actions on his behalf.

## 11.3 Recommended best practices

In this section we discuss best practices.

### 11.3.1 Server side

For the server side:

- ▶ Web 2.0 offers great flexibility to transfer state management to the browser side, but it should be done carefully. Business logic and secure data should be maintained on the server. Ajax gives you the possibility for a much more scalable and stateless middle tier. An SOA Appliance optimized for security, like Data Power, can probably help move towards that vision faster.
- ▶ Appropriate encoders and decoders should be used for JSON at the server side.
- ▶ All data received from XMLHttpRequest requests must be validated after authorization check.

**Note:** In 12.4, “JSON compression” on page 579 we discuss securing server-side resources by applying Web container security to applications.

### 11.3.2 Browser side

For the browser side:

- ▶ Input validation: This is the first step toward protecting Web applications. Input validation and sanitation filter out all possible active or malicious content from untrusted input. Two types of input validation are:
  - Blacklisting: In this approach, all characters in the blacklist are filtered out from the input. The biggest challenge with blacklisting is to ensure that all dangerous characters are listed. Since it is not possible to foresee all possible combinations of input, blacklisting often fails to validate correctly.
  - Whitelisting: This alternative lists all the allowed characters and removes all other characters from the input. The big challenge for whitelisting is to keep the list as short as possible while you still provide enough flexibility to allow the type of input required for the Web application.

You cannot consider blacklisting or whitelisting to be a foolproof solution. However, whitelisting is generally considered the more secure option. Therefore, we recommend that you use whitelisting to clean up potentially dangerous input. Escaping special characters such as changing the less than symbol (<) to "&lt;" in the string that you send to and display on browsers is

another way to improve security. Some programming languages provide useful built-in functions to escape special characters.

- ▶ **Vulnerability checking tools:** Many Web applications are vulnerable due to the similar type of programming errors in applications. Therefore, security experts have developed tools to detect these insecure programming practices. Such tools, called vulnerability checking tools, detect the potential vulnerabilities in advance by using an automated Web application vulnerability-assessment tool, which offloads the burden of checking. The tool crawls the site and then launches all the variants that it knows against all of the scripts that it found by trying the parameters, the headers, and the paths. Each input to the application (parameters of all scripts, HTTP headers, path) is checked with as many variations as possible, and if the response page contains the JavaScript code in a context where the browser can execute it, then an XSS vulnerability is exposed. One of the most common vulnerabilities that these tools detect is when programmers forget to call a sanitation routine on potential malicious input. We have discussed such tools available in section Tools 12.5 of this chapter.
- ▶ **Appropriate content-type response header:** By default, most HTTP responses generated by a Web component include a “content-type” header value of “text/html” or “text/plain”. These responses are treated by a Web browser as HTML and get loaded in the browser DOM. When rendering responses for Ajax requests, non-HTML content (like XML or JSON) is typically returned, so it is important to specify the correct “Content-Type” HTTP response header. For example, XML messages returned by Ajax calls should have a “Content-Type: text/xml” header. These responses will not be loaded into the browser DOM (based on their content-type), which can potentially thwart XSS attacks in the absence of other controls like proper output encoding.
- ▶ **The JavaScript eval function should be used carefully,** as it can execute any javascript snippet passed to it. JSON is subset of Javascript and should be encoded and decoded correctly. One approach is to use the regular expressions defined in RFC 4627 to make sure that the JSON data does not contain active parts. An even more secure alternative is using a JSON parser to parse the JSON data. Since the grammar of JSON is fairly simple, you can implement such a parser easily without any noticeable performance difference. To parse JSON you can also use `parseJSON` or `parse` methods available at <http://www.json.org/json.js> and <http://www.json.org/json2.js>, respectively. These methods refer to regular expressions to ensure a valid JSON structure. It is a good idea to parse JSON content before calling the eval function.
- ▶ **Every Javascript code loaded from different sources on your page can also access confidential user information and can try accessing the server on a user’s behalf.** Javascript is used to load libraries from different sources so you should do so only if you maintain a high level of trust with the vendor. The

client is at risk from malicious JavaScript that could be inserted in response handlers. Make sure that your browser is only executing JavaScript from a trusted server site. For example, Mozilla allows for execution of signed scripts only.

- ▶ Use <iframe> when integrating distrusted contents. You can take advantage of the same-origin policy to make it harder for attackers to get access to the full DOM tree. When you load data from a different domain into an <iframe>, give that data its own JavaScript execution context and DOM tree. This prevents attackers from stealing information from the main page. It is a good practice to use the <iframe> as much as possible to confine untrusted external contents.
- ▶ Everything that the browser sends to the server should be validated. Any confidential information being sent or received by the browser should be encrypted. Also use Secure Socket Layer (SSL) whenever required, as it provides link encryption.

## 11.4 Securing RPCAdapter (application server security)

Security support in the RPC Adapter is achieved using J2EE Web security. All the services have their own unique URLs. Access to those URLs is restricted by using J2EE Web security. This involves creating a security realm in the application server and then defining role-based access to different URLs in the deployment descriptor file, Web.xml, and then mapping these roles to users or groups in the security-realm using server-specific configuration.

**Note:** For detailed instructions about implementation of core J2EE security in WebSphere Application Server you can refer Chapter 11 of IBM Redbooks publication *Experience J2EE Using WebSphere Application Server v6.1* available from:

<http://www.redbooks.ibm.com/abstracts/sg247297.html>

The following steps assume that the RPC Adapter project was already created in 6.2.2, “Sample RPC Adapter implementation” on page 100. To secure the RPC Adapter:

1. In the Servers view start the application server and then right-click to open the administration console.
2. In the administration console select **Security** → **Secure administration, applications and infrastructure**.

- Under Application Security, select **Enable application security** and click **Apply**, as shown in Figure 11-1.

The screenshot displays the 'Security Configuration Wizard' interface. It features two tabs at the top: 'Security Configuration Wizard' (active) and 'Security Configuration Report'. The main area is divided into several sections:

- Administrative security:** Includes a checked checkbox for 'Enable administrative security' and links for 'Administrative User Roles' and 'Administrative Group Roles'.
- Application security:** Includes a checked checkbox for 'Enable application security'.
- Java 2 security:** Includes an unchecked checkbox for 'Use Java 2 security to restrict application access to local resources', and two unchecked checkboxes for 'Warn if applications are granted custom permissions' and 'Restrict access to resource authentication data'.
- User account repository:** Includes a text field for 'Current realm definition' containing 'Federated repositories', and a section for 'Available realm definitions' with a dropdown menu set to 'Federated repositories' and buttons for 'Configure' and 'Set as current'.
- Authentication:** Includes an unchecked checkbox for 'Use domain-qualified user names', and several expandable sections: 'Web security', 'RMI/IIOP security', 'Java Authentication and Authorization Service' (with a link for 'Authentication mechanisms and expiration'), 'External authorization providers', and 'Custom properties'.

Figure 11-1 Security configuration

- Save the changes in the Messages box by selecting **Save directly to Master configuration**.
- Log out and restart the server.

After restarting the server, we need to implement declarative security on the in application server and so need to define users and role.
- Run the administration console from the Servers view by right-clicking **server**.
- Expand **Users and Groups** → **Manage Groups**.

8. Add some user groups based on application roles. In our case we added three groups admins, managers, and users, as shown in Figure 11-2.

**Search for Groups**

Search by:  \* Search for:  \* Maximum results:

Group name:  \*

3 groups matched the search criteria.

Select	Group name	Description	Unique Name
<input type="checkbox"/>	<a href="#">admins</a>		cn=admins,o=defaultWIMFileBasedRealm
<input type="checkbox"/>	<a href="#">managers</a>		cn=managers,o=defaultWIMFileBasedRealm
<input type="checkbox"/>	<a href="#">users</a>		cn=users,o=defaultWIMFileBasedRealm

Page 1 of 1 Total: 3

Figure 11-2 Manager groups

9. Expand **Users and Groups** → **Manage Users**.
10. Add users for your application and adding group membership accordingly. In our case we add three users, as shown in Figure 11-3.

**Search for Users**

Search by:  \* Search for:  \* Maximum results:

User ID:  \*

4 users matched the search criteria.

Select	User ID	First name	Last name	E-mail	Unique Name
<input type="checkbox"/>	<a href="#">Ankur</a>	Ankur	Ankur		uid=Ankur,o=defaultWIMFileBasedRealm
<input type="checkbox"/>	<a href="#">goyal</a>	Ankur	Goyal		uid=goyal,o=defaultWIMFileBasedRealm
<input type="checkbox"/>	<a href="#">matt</a>	Matt	Perrins		uid=matt,o=defaultWIMFileBasedRealm
<input type="checkbox"/>	<a href="#">mike</a>	Michael	Connolly		uid=mike,o=defaultWIMFileBasedRealm

Page 1 of 1 Total: 4

Figure 11-3 Manager users

11. Log out from Administration Console and close the window.

Next we define declarative security in the Web application (RPCadapter) to have protected access to URLs/resources.

1. Open Web Deployment Descriptor from Project Explorer by expanding the **ItsorPCAdapter** project and double-clicking **Deployment Descriptor: ItsorPCAdapter**. Once Deployment Descriptor is open, switch to the security settings.
2. Add security roles USERS, MANAGERS, and ADMINS from Security Roles section.
3. Collapse Security Roles and expand **Security Constraints**. From this view you perform three separate steps:
  - a. Add the constraint.
  - b. Add the Web resource collections.
  - c. Add the authorization roles.

4. Add a security constraint to provide access to the `/httprpc/customerlist/getCustomerList` URL pattern to the managers group. Save and close the file, as shown in Figure 11-4.

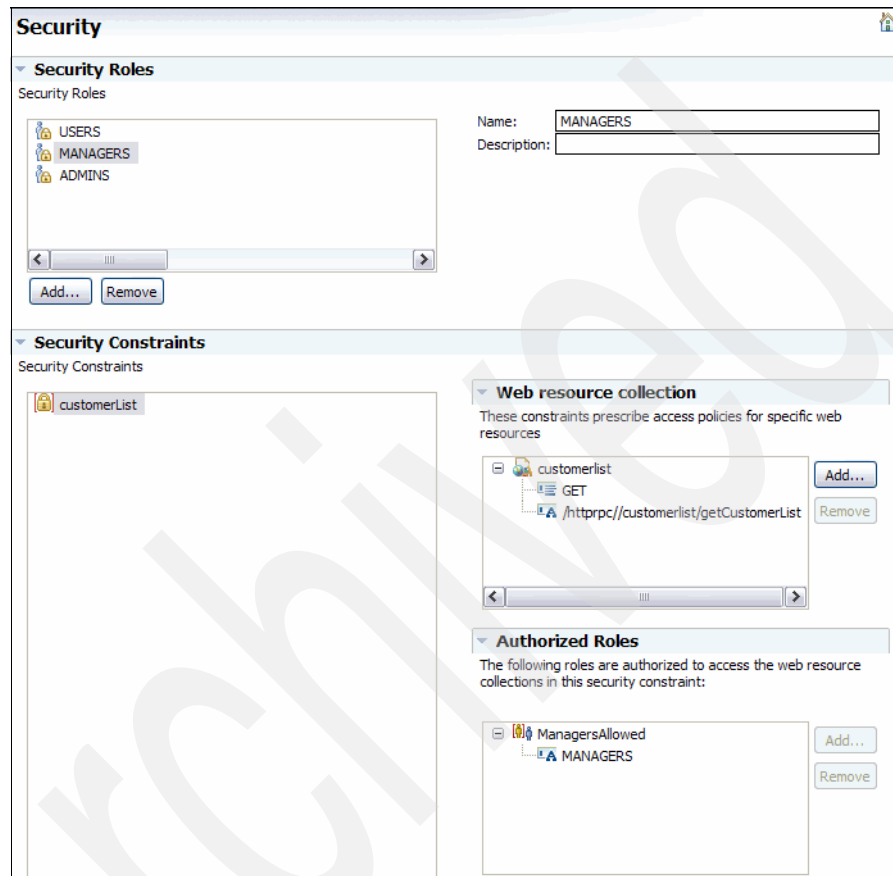


Figure 11-4 Web Deployment Descriptor: Security



5. The configuration of Web Deployment Descriptor is over. Expand **ItsoRPCAdapterEAR**, double-click **Deployment Descriptor : ItsoRPCAdapterEAR**, and switch to the **Security** tab, as shown in Figure 11-5.

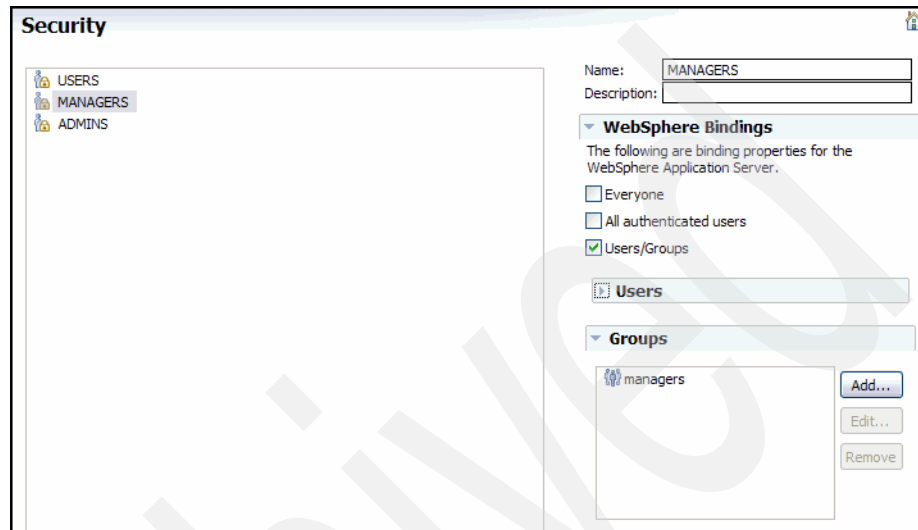


Figure 11-5 EAR Deployment Descriptor: Security

6. Click **Gather** in the Security section. This fetches all security roles defined in supporting modules.
7. We must specify WebSphere binding of security roles defined in the application to groups defined in the WebSphere Application Server. Select a security role in the Security section, select **Users/Groups** in the WebSphere Bindings section, and add a group in the Groups section, corresponding to a group defined in WebSphere application server.
8. Repeat the previous step for all security roles defined. Save the file and close.
9. Publish the application to the server and try to access the URL <https://localhost:9443/ItsoRPCAdapter/RPCAdapter/httprpc//customerlist/getCustomerList>. You will be asked for your user name and password, and only users who are members of the managers group will be allowed access.
10. Form-based authentication can also be configured for protected access to resources available on the server.

**Note:** Detailed instructions to configure form-based authentication are available in the section “Secure the Application Server Web Container” in the Redbooks publication *IBM WebSphere Application Server v6.1 Security Handbook* available from:

<http://www.redbooks.ibm.com/abstracts/sg246316.html>

## 11.5 Tools

In this section we discuss tools.

### 11.5.1 Rational Application Developer

Rational Application Developer provides a code review tool, which is a rule-based static analysis tool for automatic validation of Java code for correctness and compliance with coding standards and best practices. The rule set includes J2SE™ Security as well as J2EE security. Each rule in the code review provides an explanation of the rule, examples of violations of the rule along with solutions to correct the example violation, and, possibly, a quick fix that can be applied automatically by the application developer. Each rule can be enabled or disabled individually or by category. Users can also create their own rules from templates that are provided. A Complete Code Review has over 200 rules, and a Quick Code review has 34 of the most commonly breached rules.

The J2EE Security and J2SE Security rule categories cover potential problems for security code. Automated code review using security rules performs static analysis on code and finds violations of security rules in our Java code, as shown in Figure 11-6.

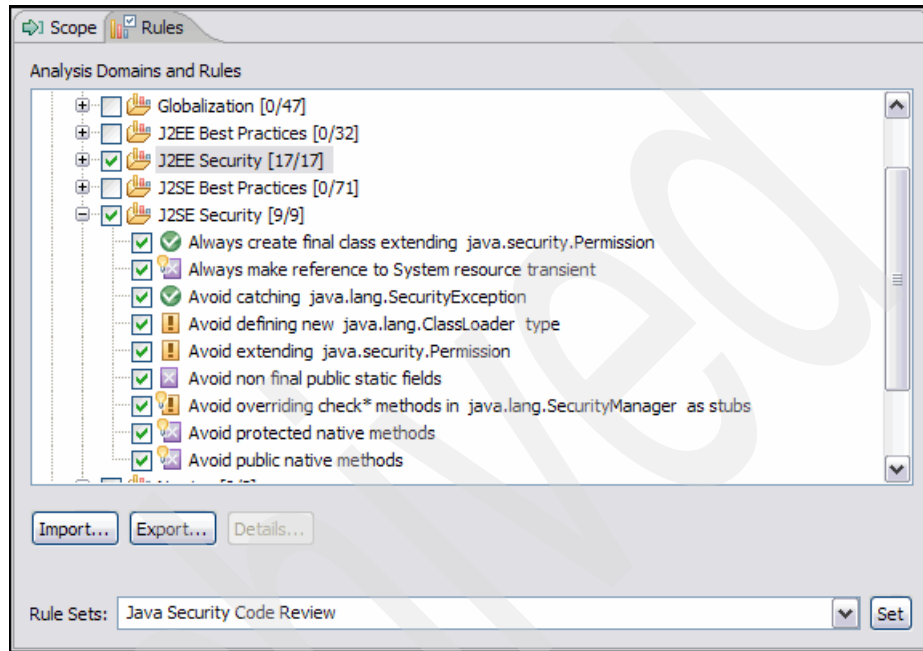


Figure 11-6 Static analysis: Code review

**Note:** For detailed instructions on configuring static analysis refer to Chapter 7, “Analyzing source code section,” in the Redbooks publication *Rational Application Developer v7 Programming Guide*:

<http://www.redbooks.ibm.com/abstracts/sg247501.html>

## 11.5.2 WebSphere DataPower XML Security Gateway XS40

With use of Ajax, people are sending more XML requests through the browser. You may experience the increased risk of XML threats. A solution might be an SOA appliance, such as DataPower®, which specializes in XML threats.

IBM WebSphere DataPower XML Security Gateway XS40 is a specialized network device that ensures complete security for XML Web services. In contrast to distributed software solutions, it provides protection with no deterioration in

performance. This appliance provides a security-enforcement point for XML and Web services transactions, including encryption, firewall filtering, digital signatures, schema validation, WS-Security, XML access control, XPath, and detailed logging.

### **XML/SOAP firewall**

IBM WebSphere DataPower XML Security Gateway XS40 filters traffic at wirespeed, based on information from layers two through seven of the protocol stack; from field-level message content and SOAP envelopes to IP address, port/host name, payload size, or other metadata. Filters can be predefined with an easy point-and-click XPath filtering GUI and automatically uploaded to change security policies based on time of day or other triggers.

### **XML/SOAP data validation**

With its unique ability to perform XML schema validation as well as message validation at wirespeed, the XS40 ensures that incoming and outgoing XML documents are legitimate and properly structured. This protects against threats such as XML Denial of Service (XDoS) attacks, buffer overflows, or vulnerabilities created by deliberately or inadvertently malformed XML documents.

### **Field-level message security**

The XS40 selectively shares information through encryption/decryption and signing/verification of entire messages or of individual XML fields. These granular and conditional security policies can be based on nearly any variable including content, IP address, host name, or other user-defined filters.

### **XML Web Services Access Control**

The XS40 supports a variety of access control mechanisms, including SAML, extensible Access Control Markup Language (XACML), SSL, LDAP, RADIUS, and simple client/URL maps. The XS40 can control access rights by rejecting unsigned messages and verifying signatures within SAML assertions.

### **Service virtualization**

XML Web Services requires companies to link partners to resources without leaking information about their location or configuration. With the combined power of URL rewriting, high-performance XSL transforms, and XML/SOAP routing, the XS40 transparently map a rich set of services to protected back-end resources with high performance.

### **Centralized policy management**

The XS40's wirespeed performance enables enterprises to centralize security functions in a single drop-in device that enhances security and reduces ongoing

maintenance costs. Simple firewall functionality can be configured via a Web GUI, and running in minutes or using the power of XSLT, the XS40 can also create sophisticated security and routing rules. Because the XS40 works with leading policy managers, it is an ideal policy execution engine for securing next-generation applications. Manageable locally or remotely, the XS40 supports SNMP, script-based configuration, and remote logging to integrate seamlessly with leading management software.

### **Web services management/service level management**

With support for Web Services Distributed Management (WSDM), Universal Description, Discovery, and Integration (UDDI), Web Services Description Language (WSDL), Dynamic Discovery, and broad support for service level management configurations, the XS40 natively offers a robust Web services Management framework for:

- ▶ The efficient management of distributed Web service endpoints and proxies in heterogeneous SOA environments
- ▶ SLM alerts and logging
- ▶ Pull and enforce policies
- ▶ Enabling broad integration support for third-party management systems and unified dashboards
- ▶ Robust support and enforcement for governance frameworks and policies

For more information about WebSphere DataPower SOA Appliances you can refer to the following Redpapers publications available at:

- ▶ Part I: Overview and Getting Started  
<http://www.redbooks.ibm.com/abstracts/redp4327.html>
- ▶ Part II: Authentication and Authorization  
<http://www.redbooks.ibm.com/abstracts/redp4364.html>
- ▶ Part III: XML Security Guide  
<http://www.redbooks.ibm.com/abstracts/redp4365.html>

## **11.5.3 Rational AppScan**

IBM Rational AppScan is an application security testing suite that scans and tests for all common Web application vulnerabilities, such as SQL-Injection, cross-site scripting, and buffer overflow. It provides broad application coverage, including Web 2.0/Ajax applications. AppScan provides intelligent fix recommendations and advanced remediation capabilities, such as comprehensive task lists necessary to fix vulnerabilities uncovered during the

scan and improve an organization's overall security posture. IBM Rational AppScan helps ensure the security and compliance of Web applications throughout the software development life cycle. IBM Rational AppScan scans Web applications, tests for security issues, and provides actionable reports and fix recommendations. IBM Rational AppScan maintains confidence in product environments by continuously auditing for known vulnerabilities and compliance reporting.

AppScan's State Inducer introduces automated scanner support for multi-step business processes, such as adding to a shopping cart and checking out, or applying for a loan, where the tester must be aware of the entire sequence as a whole. The new State Inducer enables IBM Rational AppScan to handle such server-side state, knowing when and how to reestablish it for accurate testing. The State Inducer further extends IBM Rational AppScan's support of Web 2.0 technologies following earlier success in scanning Ajax and Flash applications.

Automated analysis of all test responses for inadvertently triggered issues, new SSL tests, and Cross-Site Request Forgery (CSRF) testing are only some of the new security tests added recently.

In AppScan's test, it will use a new session to verify the CSRF vulnerability. There have been many questions as to why AppScan uses a new session when the actual exploit would use the same session that the logged in user has. The test request using a new session is for accuracy reasons.

AppScan identifies a meaningful request to test for CSRF. The request using a new session allows AppScan to identify a meaningful request that has session identifiers in cookies only.

If AppScan is able to get the same response to the meaningful request in the new session it can be sure that the session identifiers are in cookie parameters only, and that the request is vulnerable to CSRF exploits.

Rational AppScan Developer Edition software extends Rational Application Developer for WebSphere with support for Web application security testing that automates vulnerability assessments. Rational AppScan Developer Edition is an offering created to empower developers with the ability to invoke Web application security testing within their development environment in a nondisruptive fashion.

## Best practices

This chapter describes a set of best practices that should be adhered to when designing the Dynamic Ajax application with WebSphere Application Server.

This chapter describes the following:

- ▶ ShrinkSafe
- ▶ JavaServer™ Pages for JSON and JSON compression
- ▶ Internationalization (i18n)
- ▶ Accessibility (a11y)

## 12.1 Introduction

This chapter describes a set of very different technologies, best practices, and information that is considered important for developers who will be creating applications that implement functions included with the Web 2.0 Feature Pack. This chapter will help you to prepare your applications for enterprise class deployment into production run times of the WebSphere Application Server.

## 12.2 ShrinkSafe and the Dojo Build System

ShrinkSafe is a JavaScript *compression* system. It can typically reduce the size of your scripts by a third or more. It is a tool that has been developed by the Dojo Toolkit team for help with packaging and preparing Dojo content for production deployment.

**Note:** The size of the compression can depend on your programming style.

There are a variety of other tools that also shrink JavaScript files, but ShrinkSafe is different. Instead of relying on brittle regular expressions, ShrinkSafe is based on the Apache Rhino JavaScript interpreter. This allows ShrinkSafe to transform the source of a file with much more confidence that the resulting script will function identically to the file that you uploaded. ShrinkSafe does not change any public variables or APIs, thereby keeping compressed code consistent with the original source. That means that you can drop the compressed version of your JavaScript into your pages without changing the code that uses it.

### 12.2.1 Why ShrinkSafe

If your Web applications do not include significant amounts of JavaScript, then you will not need to use ShrinkSafe. If, on the other hand, you are really building an amazing Web 2.0 user experience, ShrinkSafe can make your pages respond faster by reducing the number of HTTP requests needed as well as decreasing the size of the files being served. Dojo is a very modular system, meaning that each component is delivered in a separate JavaScript file. This can significantly increase the bandwidth requirements of your application. The Dojo bootstrap auto loads all the required elements of the application. If you have a very rich user interface that uses a wide variety of components then each module will be individually loaded. This behavior may be desirable during testing and debugging, but within a production deployment you need to optimize the time being spent loading the required elements.



Perceived speed is real speed for Web applications, and ShrinkSafe helps make your Web applications perform better without the effort required in re-engineering the source code.

As shown in Figure 12-1, ShrinkSafe displays all the individual modules loaded when the page is requested.

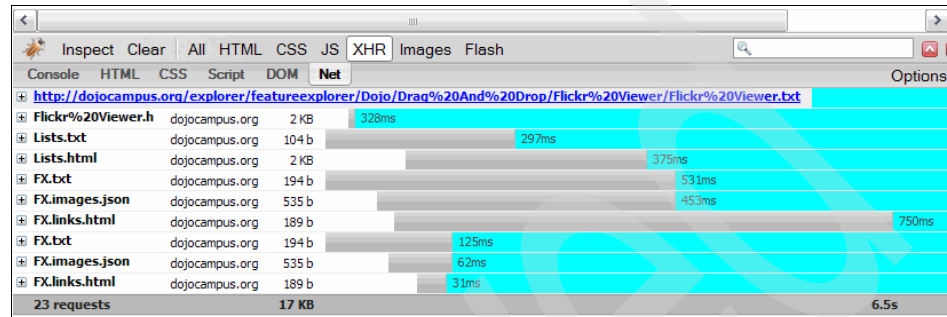


Figure 12-1 Example of the multiple requests generated when loading Dojo

## Getting started

To get started you need to download two items from the Dojo Toolkit Web site. This book is written to the Version 1.0 release of the Web 2.0 Feature Pack. This release contains the Ajax runtime, which is based upon the Dojo Toolkit 1.0.1 release of the Ajax library. Normally, you would want to keep the ShrinkSafe elements the same as the Web 2.0 Feature Pack release. The issue is that the packaging of the older ShrinkSafe elements has changed, so you should download the latest release of ShrinkSafe.

You can download the ShrinkSafe library from the Dojo Web site:

<http://download.dojotoolkit.org/current-stable/dojo-release-1.1.0-shrink-safe.zip>

When you unzip this library you will see that it contains a jar file called `custom_rhino.jar`. This file is a repackaging of the Apache Rhino project and requires a Java 1.4-compliant JVM to run.

The second file needed is the buildscripts for the 1.0.1 release. These will enable you to recreate a new layered build process that will generate a single JavaScript file that can be deployed into your application.

We now demonstrate how to use ShrinkSafe to compress a single file or be used in the Dojo Build System.

## 12.2.2 Compressing a single file

The following steps show you how to use Rational Application Developer tooling to convert JavaScript files into compressed format using the ShrinkSafe library.

1. Open up your RAD toolkit and select **File** → **New** → **Project**.
2. Select **Java Project** from the list of projects and select **Next**, as shown in Figure 12-2.

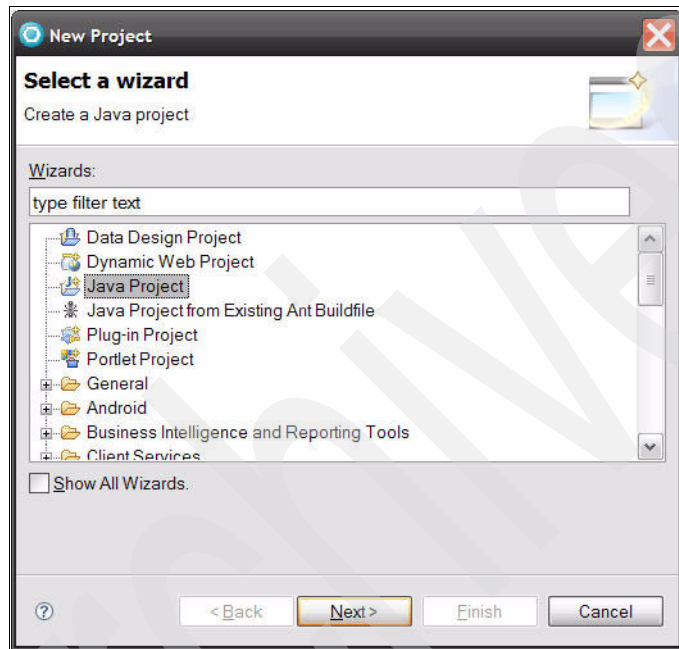


Figure 12-2 Create a new Java project to hold the JavaScript source code

3. Enter ITSOShrinkSafe as the project name and make sure that a valid Java run time is selected. In this example J2SE 5.0 is being used. Then select **Next**, as shown in Figure 12-3.

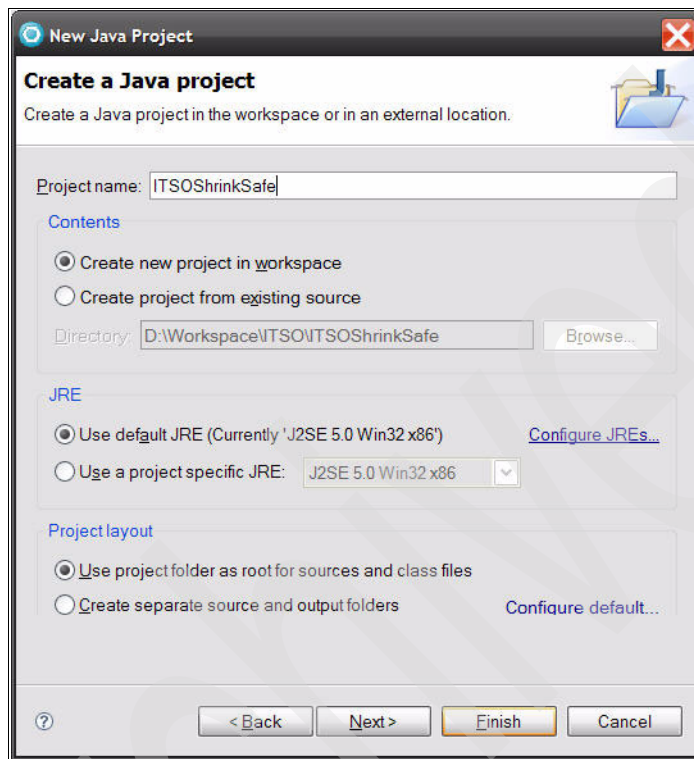


Figure 12-3 Enter the project name for the project

4. Confirm that you are creating the correct project and then click **Finish**. See Figure 12-4.

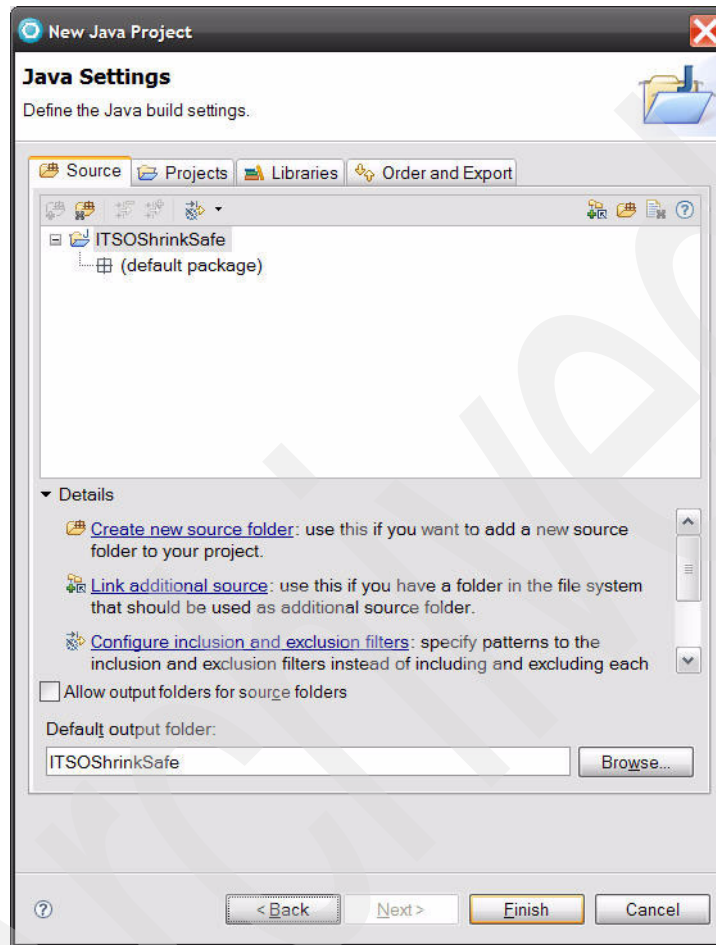


Figure 12-4 Review the settings on the Java Settings page of the create project wizard

5. With the project created we need to create an *in* and *out* source directory that will be used to hold the JavaScript source before and after conversion.
6. Select **File** → **New** → **Source Folder**.

7. Enter the name of `in` for your source folder, as shown in Figure 12-5.

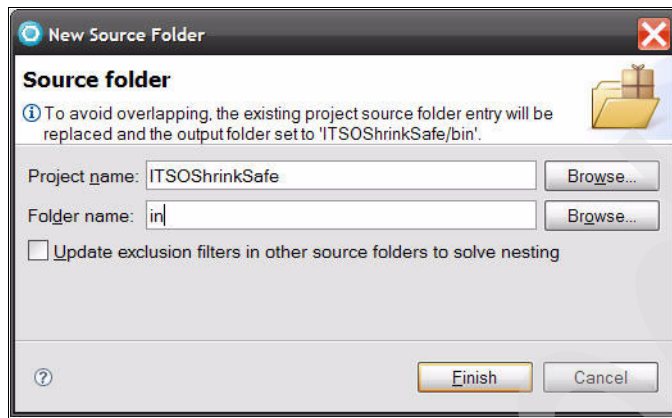


Figure 12-5 In source folder

8. Repeat the previous step for the out folder.
9. We are now ready to add the custom-built Rhino package that was downloaded earlier from the Dojo Web site. Unzip the file `dojo-release-1.1.0-shrinksafe.zip` and drag the file **custom\_rhino.jar** from the unzip directory to the root of the ITSOShrinkSafe project.
10. Open the Properties dialog for ITSOShrinkSafe and select **Java Build Path** and then the **Libraries** tab, as shown in Figure 12-6.

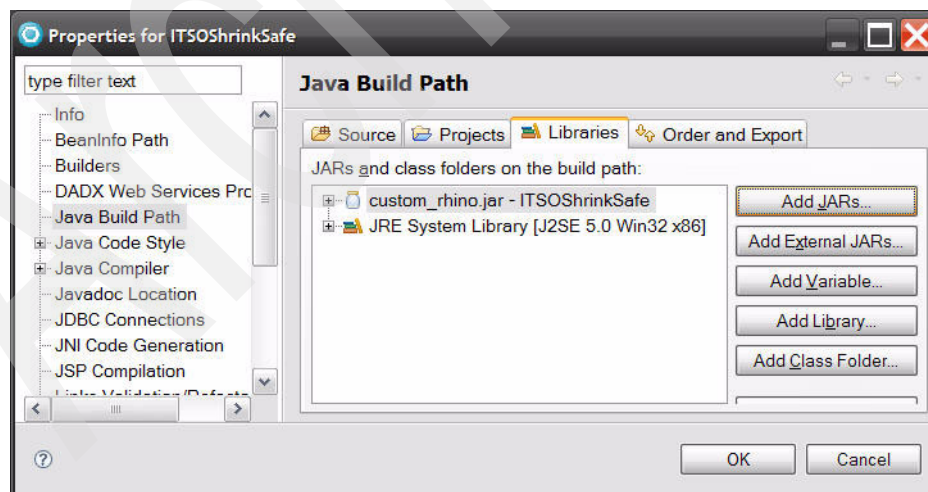


Figure 12-6 Java Build Path properties page for project

11. Click **Add JARs**, select the **custom\_rhino.jar** file, and click **OK**.

12. Click **OK** on the properties dialog to save the changes.

With the project now configured the next steps involve creating a run configuration that will run the Java Compression tool. However, before that we need to create a sample to work with.

13. Create a file named `Sample.js` in the `in` directory and add the JavaScript snippet given in Example 12-1 to it.

*Example 12-1 Sample.js to demonstrate JavaScript compression*

---

```
function MyClass(){
    this.foo = function(argument1, argument2){
        var addedArgs = parseInt(argument1)+parseInt(argument2);
        return addedArgs;
    }
    var anonymousInnerFunction = function(){
        // do stuff here!
    }
}
function MyFunc(){
    // this is a top-level function
}
// we've got multiple lines of whitespace
here
```

---

14. Save the file.

In order to run applications within Rational Application Developer or Eclipse development environments you first need to configure a run configuration. This can be done by following these simple steps:

1. From the menu in Rational Application Developer select **Run** → **Run** to open the Run configuration dialog.
2. Select **Java Application** from the list of possible run types. Using the context menu or the new launch configuration button on the toolbar create a new configuration for the Java Application type.
3. Select the **ITSOShrinkSafe** project within the Project section.
4. Select the **Search** button for the Main class section.

5. Enter `org.mozilla.javascript.tools.shell.Main` in the Select type field at the top of the dialog. Then select **OK**, as shown in Figure 12-7.

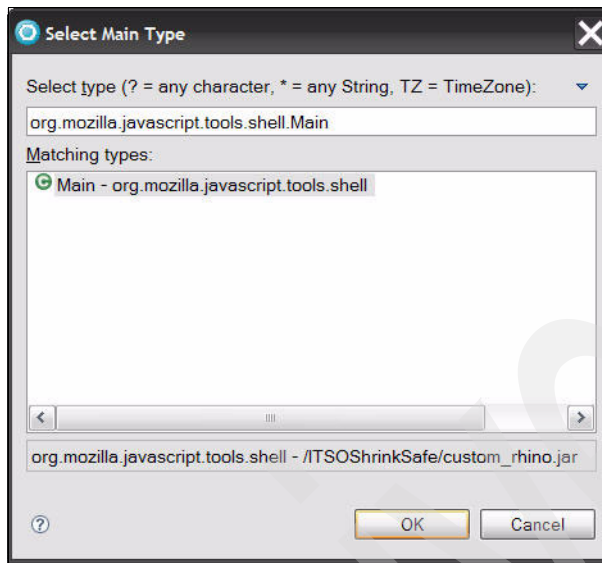


Figure 12-7 Select the Main class to run

You should now have a configuration similar to the one shown in Figure 12-8.

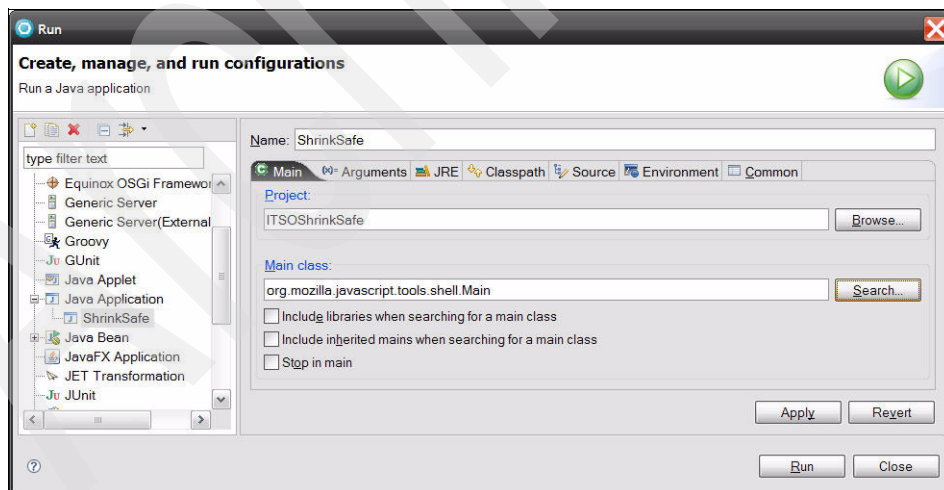
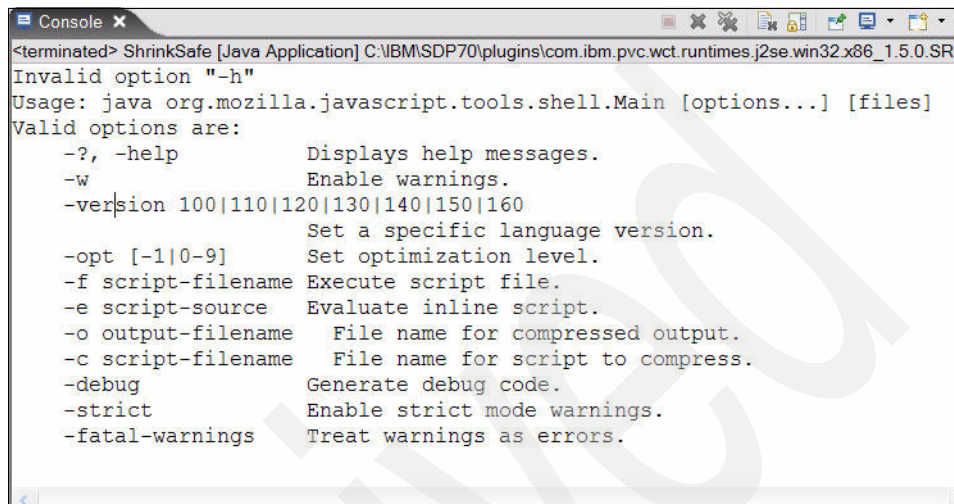


Figure 12-8 Run configuration for Rhino classes

6. Next check the configuration parameters for the main class by switching to the **Arguments** tab and entering `-h` into the Program arguments section.

7. Click **Apply** and then click the **Run** button.
8. Open the Console view to see the list of parameters that the ShrinkSafe java library supports. See Figure 12-9.



```
<terminated> ShrinkSafe [Java Application] C:\IBM\SDP70\plugins\com.ibm.pvc.wct.runtimes.j2se.win32.x86_1.5.0.SR
Invalid option "-h"
Usage: java org.mozilla.javascript.tools.shell.Main [options...] [files]
Valid options are:
    -?, -help           Displays help messages.
    -w                  Enable warnings.
    -version 100|110|120|130|140|150|160
                        Set a specific language version.
    -opt [-1|0-9]       Set optimization level.
    -f script-filename  Execute script file.
    -e script-source    Evaluate inline script.
    -o output-filename  File name for compressed output.
    -c script-filename  File name for script to compress.
    -debug              Generate debug code.
    -strict             Enable strict mode warnings.
    -fatal-warnings     Treat warnings as errors.
```

Figure 12-9 Options for the ShrinkSafe command-line tool

Notice that you can specify `-o` and `-c` for the output and input options.

9. Open the ShrinkSafe run configuration and add the follow arguments to the configurations in the Run Configuration dialog in the Program arguments section:

```
-o out\Sample.js -c in\Sample.js
```

These arguments will output a new compressed version of the Sample.js JavaScript file into the `out` directory and will receive the source from the `in` directory.

**Note:** You must specify the `-o` parameter before the `-c` parameter, otherwise the command options will not work.

10. Click **Apply** and then **Run**.
11. You should now see the following message on the console. Press the F5 key to refresh in the Package Explorer view to update the project tree.  
  
Compressed file stored in 'out\Sample.js'
12. Open the file `out\Sample.js` in a text editor in the tooling.



Notice that the Sample.js JavaScript file has been compressed and the whitespace and the long variable and method names have been replaced as shown in Example 12-2.

*Example 12-2 Compressed version of the Sample.js file*

---

```
function MyClass(){
this.foo=function(_1,_2){
var _3=parseInt(_1)+parseInt(_2);
return _3;
};
var _4=function(){
};
};
function MyFunc(){
};
```

---

With this example you have seen how the ShrinkSafe package can be integrated into the Rational Application Developer based tooling and then used to compress a simple JavaScript file. The following section explains how to use the Dojo build system to create a custom build.

### 12.2.3 Dojo build and layering system

For small JavaScript projects placing the JavaScript code in a single file makes perfect sense. However, this can add significant application overhead when working with a larger JavaScript framework since every .js file must be loaded into the browser for consumption by the application. For this reason the Dojo team developed a building and packaging system that is built around ShrinkSafe. The Dojo custom build speeds performance by doing the following:

1. It groups modules together into layers. A layer, which is one large .js file, loads faster than if the .js modules that comprise it are loaded individually.
2. It interns external non-JavaScript files. This is the most important for Dijit templates, which are kept in a separate HTML file. Interning pulls the entire file in and assigns it to a string.
3. It smooshes the layer down with ShrinkSafe. ShrinkSafe removes unneeded whitespace and comments and compacts variable names down to smaller ones. This smooshed file downloads and parses faster than the original.
4. It copies all non-layered scripts to the appropriate destination. While this does not decrease the time spent loading, it does ensure that all Dojo modules can be loaded, even if not present in a layer. If you use a particular module only once or twice, keeping it out of the layers makes those layers load faster.

Next you must designate the modules in each layer with a profile, which is something like a Makefile or Ant script. The profile is defined in JSON format.

The input to the build system is the Dojo source tree, any source trees for custom widgets that you wish to include, and the profile. The output is a Dojo distribution tree that you can copy to your Web server.

### ***Prerequisites***

Prior to running the Dojo's build system you need the Java 1.4.2 or later (Java 1.5 recommended) installed, which is a source build of Dojo, which you can obtain at:

<http://download.dojotoolkit.org/>

Notice that the source builds are suffixed with -src.

The dependencies section within the layer lists all the modules that are called directly. Referenced modules are also included, so you do not have to trace back the dependency tree. Also, Dojo base modules are an implicit dependency, so you do not need to list things like "dojo.query". (Dojo core modules, however, do need to be listed.)

The modules for that layer are gathered together to make the layer file. In the example described later we first created an "itsodojo.js" layer and then loaded this layer file using a SCRIPT tag:

```
<script type="text/javascript" src="js/itsodojo.js" djConfig="isDebug:
true, parseOnLoad: true"></script>
```

## **12.2.4 Creating a custom build**

To create a custom build of a Dojo layer for your project you need to download a complete source package from the Dojo Web site. The Web 2.0 Feature pack does not include the build packaging system. Download the code for the relevant version of Dojo that is packaged into the Web 2.0 Feature Pack.

To download the buildscripts from the 1.0.1 release directory select the **dojo-release-1.0.1-src.zip** file. This contains the buildscripts and matches the new process for the 1.1.0 release. See Figure 12-10.

<http://download.dojotoolkit.org/release-1.0.1/>

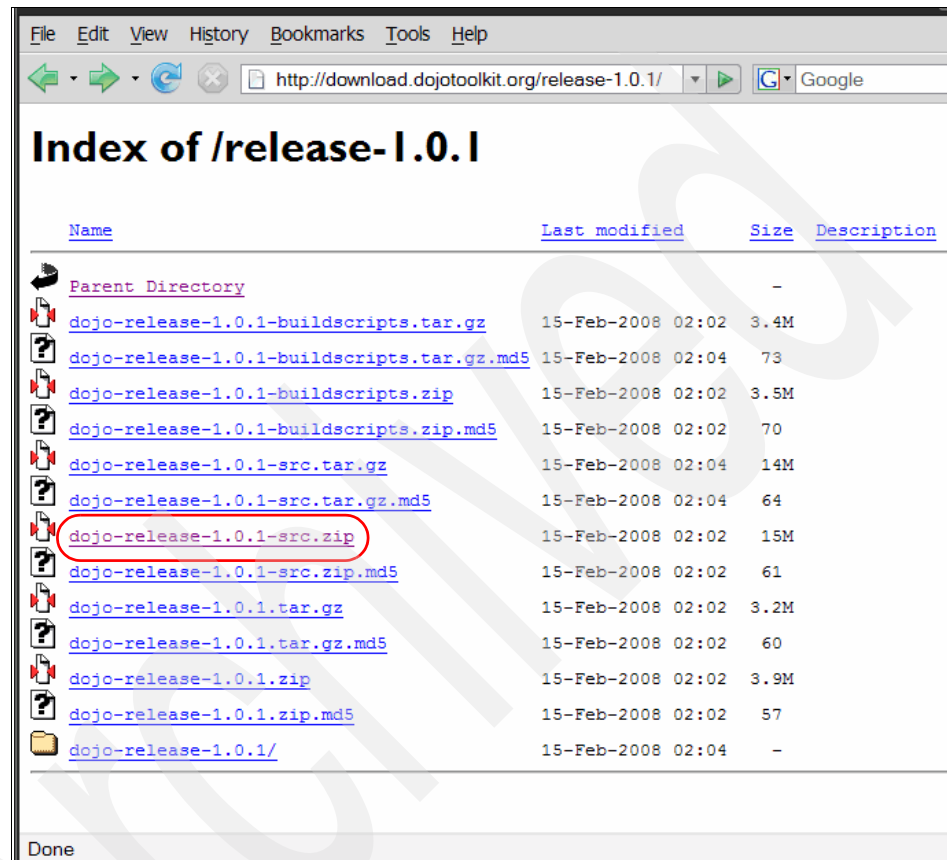


Figure 12-10 Download the `dojo-release-1.0.1-src.zip` file

The following steps will walk you through the creation of a custom build process for an application:

1. Open the RAD tooling and create a new Java project named ITSODOjoCustomBuild.
2. Within the project, create a new directory called build that will contain the Dojo source structure.
3. Select the **build** folder, select **File** → **Import**, and then select **Archive file** from the list of import choices.

4. Select the dojo source zip file that you downloaded earlier. It should be dojo-release-1.0.1-src.zip. See Figure 12-11.

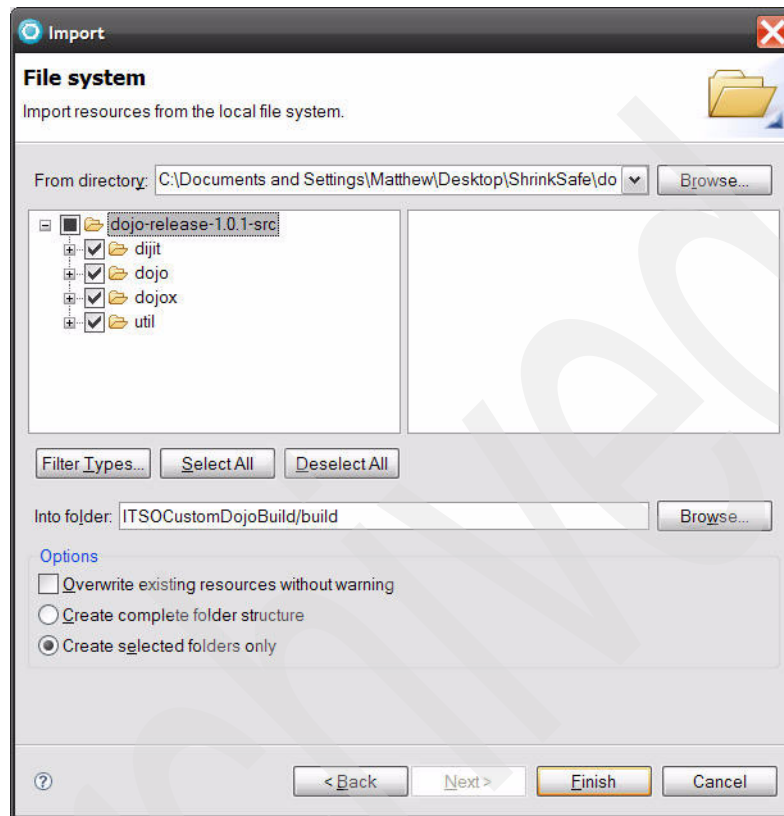


Figure 12-11 Import Dojo Source into your Custom build project

5. Select all the folders in the Dojo source zip file and ensure that the Into folder name is specified as ITSOCustomDojoBuild/build.
6. Click **Finish** to complete the import of the Dojo source code.

7. The next step is to create a custom layer for the build. If you expand the `dojo-release-1.0.1.src/util/buildscripts/profiles` directory you will see a list of the profiles that are supplied within the source package, as shown in Figure 12-12.

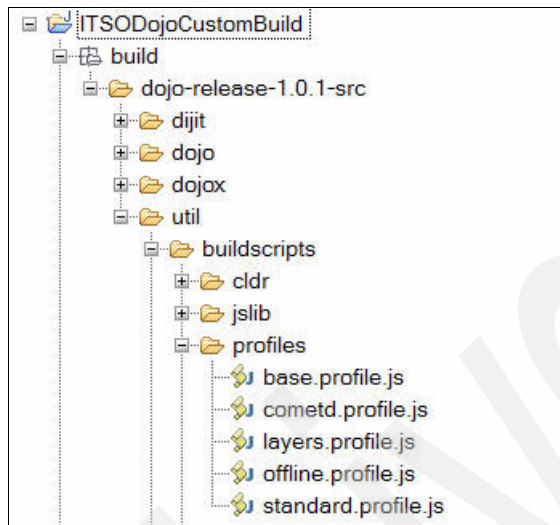


Figure 12-12 List of supplied profiles within the Dojo Source package

8. Within the profiles directory create a new profile called `itso.profile.js`.
9. Add the snippet shown in Example 12-3 to the `itso.profile.js` file.

Example 12-3 Sample `itso.profile.js` file for the dojo build system

```
dependencies = {
  layers: [
    {
      name: "dojo.js",
      dependencies: [
        "dojo.*",
        "dijit._Templated",
        "dijit._Widget",
        "dijit.form.Button",
        "dijit.form.FilteringSelect",
        "dijit.form.Slider",
        "dojo.data.ItemFileReadStore",
        "dojo.data.util.filter",
        "dojo.data.util.simpleFetch",
        "dojo.date.stamp",
        "dojox.collections.ArrayList",
```

```
        "dojox.collections.Dictionary",
        "dijit._Container",

    ]
}
],
prefixes: [
    [ "dijit", "../dijit" ],
    [ "dojox", "../dojox" ],
]
};
```

---

Observe that this profile is going to build a layer of Dojo that only includes specific dojo,dijit and dojox package elements. To build an optimum list you should examine your project to determine which components are being used.

With the profile now specified the final step is to create the run external configuration so that the build process can be completed.

10. Select **Run** → **External Tools** → **External Tools**. This opens up the External tool configuration dialog.

11. Click **Program** and create a new configuration called Dojo Custom Build, as shown in Figure 12-13.

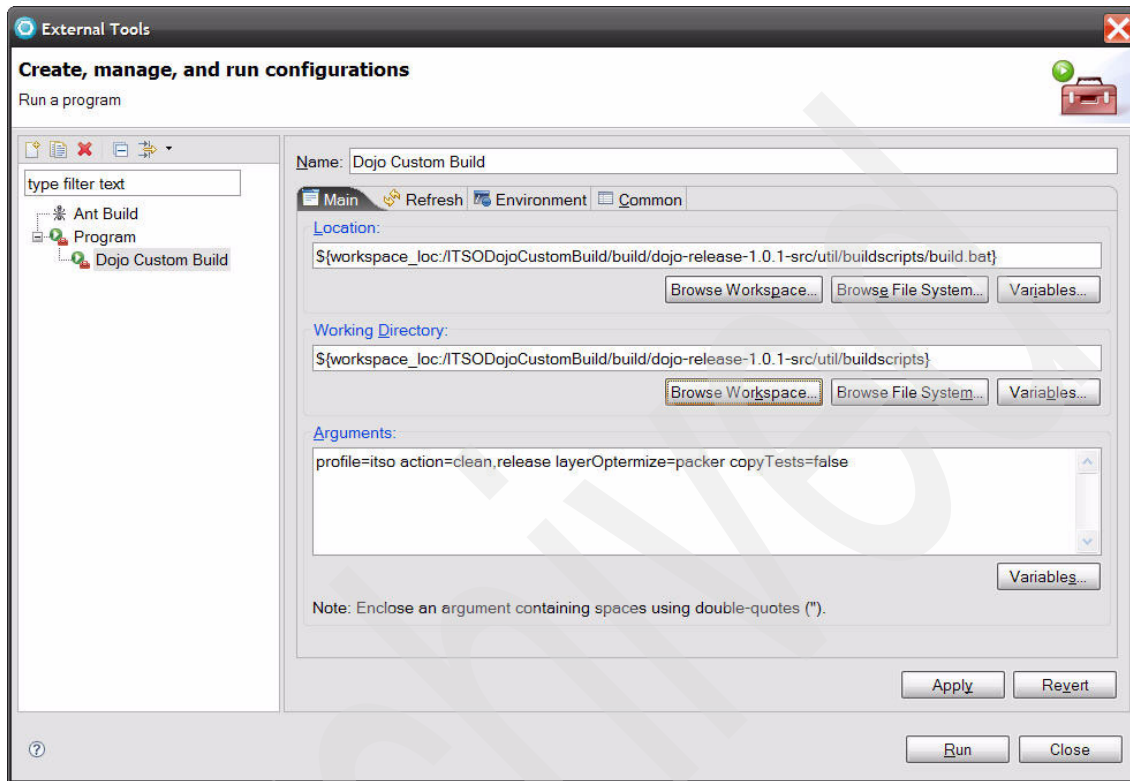


Figure 12-13 External tools configuration for the custom Dojo build

12. Using **Browse Workspace** on the Location field select the **build.bat** script file from the ITSODojoCustomBuild/build/dojo-release-1.0.1-src/util/buildscripts directory.
13. Using the **Browse Workspace** button on the Working Directory field select the buildscripts as the working directory (that is, ITSODojoCustomBuild/build/dojo-release-1.0.1-src/util/buildscripts).

14. The final step is to configure the arguments for the external tool. The build tool supports the parameters listed in Table 12-1.

Table 12-1 Parameters supported by the Dojo build system

Name	Description	Default
profile	The name of the profile to use for the build. It must be the first part of the profile file name in the profiles/ directory. For instance, to use base.profile.js, specify profile=base.	base
profileFile	A file path to the profile file. Use this if your profile is outside of the profiles directory. Do not specify the "profile" build option if you use "profileFile".	""
action	The build actions to run. Can be a comma-separated list, like action=clean,release. The possible build actions are: <ul style="list-style-type: none"> <li>► clean</li> <li>► release</li> </ul>	"help"
version	The build will be stamped with this version string.	"0.0.0.dev"
localeList	The set of locales to use when flattening i18n bundles.	"en-gb,en-us,de-de,es-es,fr-fr,it-it,pt-br,ko-kr,zh-tw,zh-cn,ja-jp"
releaseName	The name of the release. A directory inside 'releaseDir' will be created with this name.	"dojo"
releaseDir	The top-level release directory where builds end up. The 'releaseName' directories will be placed inside this directory.	".././release/"
loader	The type of dojo loader to use. "default" or "xdomain" are acceptable values.	"default"



Name	Description	Default
internStrings	Turn on or off widget template/ <code>dojo.uri.cache()</code> file interning.	true
optimize	Specifies how to optimize module files. If "comments" is specified, then code comments are stripped. If "shrinksafe" is specified, then the Dojo compressor will be used on the files, and line returns will be removed. If "shrinksafe.keepLines" is specified, then the Dojo compressor will be used on the files, and line returns will be preserved. If "packer" is specified, Then Dean Edwards' Packer will be used.	""
ayerOptimize	Specifies how to optimize the layer files. If "comments" is specified, then code comments are stripped. If "shrinksafe" is specified, then the Dojo compressor will be used on the files and line returns will be removed. If "shrinksafe.keepLines" is specified, then the Dojo compressor will be used on the layer files, and line returns will be preserved. If "packer" is specified, then Dean Edwards' Packer will be used.	"shrinksafe"
copyTests	Turn on or off copying of the test files.	true
log	Sets the logging verbosity. See <code>jslib/logger.js</code> for possible integer values	logger.TRACE

Name	Description	Default
xdDojoPath	If the loader=xdomain build option is used, then the value of this option will be used for the path to Dojo modules. The dijit and dojox paths will be assumed to be siblings of this path. The xdDojoPath should end in '/dojo'.	""

15. Add the following arguments to the External Tool configuration.

```
profile=itso action=clean,release layerOptermize=packer
copyTests=false
```

You can see in this example that we have specified the itso profile and have set the layerOptermize to packer. We have also decided not to include the tests.

16. To save this configuration select **Apply**.

17. Click the **Run** button to run the custom build process.

The console view should display output similar to Example 12-4.

*Example 12-4 Console view after a custom build has been processed*

```
D:\Workspace\ITS0\ITS0DojoCustomBuild\build\dojo-release-1.0.1-src\util
\buildscripts>java -jar ../shrinksafe/custom_rhino.jar build.js
profile=itso action=clean,release layerOptermize=packer copyTests=false
clean: Deleting: ../../release/dojo
release: Using profile: profiles/itso.profile.js
release: Using version number: 0.0.0.dev for the release.
release: Deleting: ../../release/dojo
release: Copying: ../../dojo/../../dijit to: ../../release/dojo/dijit
release: Copying: ../../dojo/../../dojox to: ../../release/dojo/dojox
release: Copying: ../../dojo to: ../../release/dojo/dojo
release: Building dojo.js and layer files
release: Interning strings for file: ../../release/dojo/dojo/dojo.js
release: Interning strings for :
../../release/dojo/dojo/dojo.js.uncompressed.js
release:      ../../release/dojo/dijit/form/templates/Button.html
release:
../../release/dojo/dijit/form/templates/DropDownButton.html
release:      ../../release/dojo/dijit/form/templates/ComboButton.html
release:      ../../release/dojo/dijit/form/templates/TextBox.html
release:      ../../release/dojo/dijit/templates/Tooltip.html
```

```
release:
../../release/dojo/dijit/form/templates/ValidationTextBox.html
release:      ../../release/dojo/dijit/form/templates/ComboBox.html
release:
../../release/dojo/dijit/form/templates/HorizontalSlider.html
release:
../../release/dojo/dijit/form/templates/VerticalSlider.html
release: Optimizing (shrinksafe) file: ../../release/dojo/dojo/dojo.js
release: Files baked into this build:

dojo.js:
./jslib/dojoGuardStart.jsfrag
../../dojo/_base/_loader/bootstrap.js
../../dojo/_base/_loader/loader.js
../../dojo/_base/_loader/hostenv_browser.js
./jslib/dojoGuardEnd.jsfrag
../../release/dojo/dojo/_base/lang.js
../../release/dojo/dojo/_base/declare.js

<lots of messages>

release: Interning strings for: ../../release/dojo/dojo
release: Build is in directory: ../../release/dojo
Build time: 35.187 seconds
```

---

The build of the Dojo layer has been performed into the release directory. If you refresh your workspace you will see that the directory has been created.

18. Open the /release/dojo/dojo/dojo.js file using Notepad and observe the results of the build process, as shown in Figure 12-14.



```
/*
Copyright (c) 2004-2007, The Dojo Foundation
All Rights Reserved.

Licensed under the Academic Free License version 2.1 or above OR the
modified BSD license. For more information on Dojo licensing, see:
http://dojotoolkit.org/book/dojo-book-0-9/introduction/licensing.html

*/

/*
This is a compiled version of dojo, built for deployment and not for
development. To get an editable version, please visit:
http://dojotoolkit.org

for documentation and information on getting the source.
*/

if(typeof dojo=="undefined"){(function(){if(typeof this["djconfig"]=="unde
p://dojotoolkit.org",d.version.toString());dojo._mixin=function(_9,_a){va
ar d=dojo;dojo.mixin(dojo,{_loadedModules:{},_inFlightCount:0,_hasResource
d["eval"](_2f+"\r\n//@ sourceURL="+uri);if(cb){cb(_30);return true;};dojo
_postLoad){return;};if(d._inFlightCount>0){console.debug("files still in f
d after loading "+_44+"");};return _43;};dojo.provide=function(_47){_47=
{continue;};var _5d=new d._Url(_a[1]+"");var _5e=new d._Url(uri+"");if(_5c
||!r[8]?":n";if(this.authority!=n){r=this.authority.match(ire);this.user
=djconfig["baseurl"];var n=navigator;var dua=n.userAgent;var dav=n.appVer
g.ieforceactivexhr){try{_77=new XMLHttpRequest();catch(e){}}if(!_77){for
5=(e&&e.type)?e.type.toLowerCase():"load";if(arguments.callee.initialized
nbeforeunload",function(){_w.setTimeout(function(){_8b=false;},0);});_87(
turn _8e(it);};_8e;}});dojo.isobject=function(it){return it!=undefined&&
"} is null (scope="_9f,"")").join("");return function(){return _9f[_
o._base,declare]=true;dojo.provide("dojo._base.declare");dojo.declare=fur
ss:bp,mixin:mp,extend:dojo.declare._extend;};if(_ba){_be.prototype=dojo._c
eof _ca.constructor){return p;};if(m&&(m=m._findMixin(Ca))){return m;};c-
_listener=getDispatcher(function(){return function(){var ap=Array.prototy
y(this,_fd);delete _fd[0];};dojo.disconnect=function(obj,_ff,_100,_101){
celler){err=this.canceller(this);else{this.silentlyCancelled=true;};if(th
llbacks(_119,null);};addErrback:function(cb,cbfn){var _11c=dojo.hitch(cb,c
o._escapeString=function(str){return (""+str.replace(/[/\|\\"']/g,"\\$1")+
=function(){return null;};var _133=[];for(var key in it){var _135;if(type
<1;)+){_p[2].call(_p[1],arr[i],i,arr);};_everyOrSome=function(_14a,arr,
_,white:[255,255,255],maroon:[128,0,0],red:[255,0,0],purple:[128,0,128],fuc
dojo.blendColors=function(_174,end,_176,obj){var d=dojo,t=obj;new dojo.co
resource["dojo._base"]=true;dojo.provide("dojo._base");(function(){if(djcor
```

Figure 12-14 Sample view of Dojo.js after the build process has completed

You can see that by using the packer the Dojo source has been tightly compressed into a single Dojo.js file.

This file is all that you need to include in your project for distribution with your application.

## Summary

What you have seen in this section is how you can use the external tools that are built around Dojo to help prepare your Dojo application for production deployment. Using the Dojo build and packaging system will become an important part of your production build process and make your application ready for the high-volume usage of most WebSphere solutions. By reducing the size

and the number of calls that the browser needs to make to the server should dramatically improve the load and run time of your application.

The sample build that was performed on the **itso.profile.js** profile produced a version of the `dojo.js` file that was significantly reduced. See Figure 12-15.

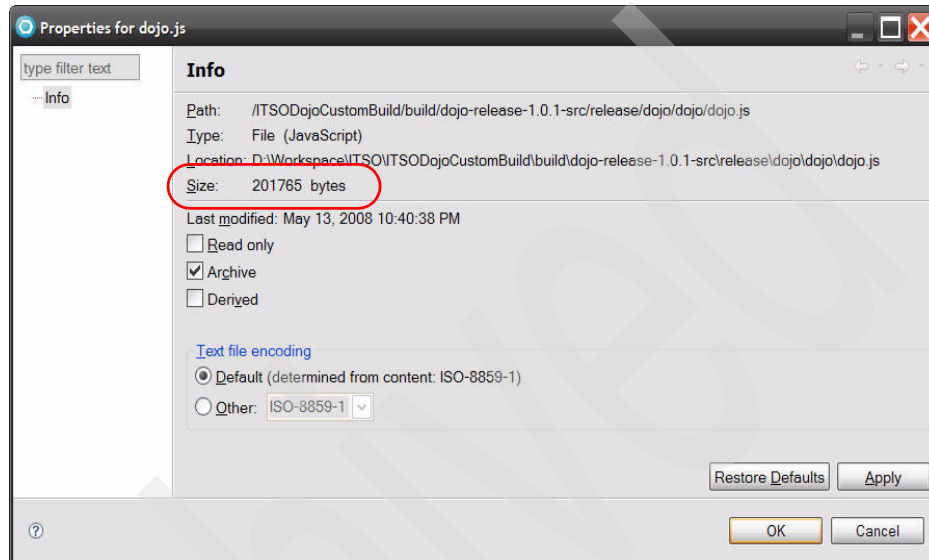


Figure 12-15 Size for the `dojo.js` produced by the build system

## 12.3 JavaServer Pages for JSON

The J2EE Specification contains a powerful templating run time called JavaServer Pages. JSPs, as they are commonly known, are very good for creating data that can be consumed by Ajax clients. One of the techniques that you may use within your project is to reuse an existing JSP page that was used to build an HTML page on the server, and modify it or duplicate it to support the generation of JSP format content. This enables you to have your existing application running while at the same time adding new function.

This technique has been successfully used on a variety of projects and is one of the ways that you can leverage the existing assets that you have within your J2EE projects.

### JSON tag library

The JSON-taglib is a JSP 2.0 tag library used to render JavaScript Object Notation (JSON) data from within JSP code. It can be used as part of the

server-side of an Ajax application, allowing you to use the full power of JSP to format your JSON data. You can download the taglib library from:

<http://json-taglib.sourceforge.net/>

To get started just drop the json-taglib.jar file into the WEB-INF/lib directory of your Web-application.

A simple example of how the taglib could be used with an Ajax e-commerce shopping cart is shown in Example 12-5.

---

*Example 12-5 Example of using the JSON tag library*

---

```
<%@ taglib prefix="json" uri="http://www.atg.com/taglibs/json" %>

<json:object>
  <json:property name="itemCount" value="${cart.itemCount}"/>
  <json:property name="subtotal" value="${cart.subtotal}"/>
  <json:array name="items" var="item" items="${cart.lineItems}">
    <json:object>
      <json:property name="title" value="${item.title}"/>
      <json:property name="description" value="${item.description}"/>
      <json:property name="imageUrl" value="${item.imageUrl}"/>
      <json:property name="price" value="${item.price}"/>
      <json:property name="qty" value="${item.qty}"/>
    </json:object>
  </json:array>
</json:object>
```

---

This JSP produces the output shown in Example 12-6.

---

*Example 12-6 JSP output*

---

```
{
  itemCount: 2,
  subtotal: "$15.50",
  items: [
    {
      title: "The Big Book of Foo",
      description: "Bestselling book of Foo by A.N. Other",
      imageUrl: "/images/books/12345.gif",
      price: "$10.00",
      qty: 1
    },
    {
      title: "Javascript Pocket Reference",
```

```
        description: "Handy pocket-sized reference for the Javascript
language",
        imageUrl: "/images/books/56789.gif",
        price: "$5.50",
        qty: 1
    }
]
}
```

---

You can see that by taking advantage of the existing J2EE programming model it is possible to produce the feeds and JSON formats that you need to link into the Web 2.0 programming model with minimal development effort.

## 12.4 JSON compression

One issue that many Web applications projects encounter is the use of very verbose value name pairs within JSON structures. Due to the simplicity of the JSON format and how easy it is to create JSON content you can easily wind up making it very verbose.

Verbose JSON is not a problem when working with small snippets that flow back to the client. The problem arises when sending a large selection of data to the client. As an example, you might need to populate a browser-based data store. This store can be queried by the client and then used to produce a unique set of keys to be sent back to the server. If this store contains 300 records and you also use a very verbose format then the data can be returned from the server to the client and can add up to a large payload.

## 12.5 Internationalization (i18n)

For a product to have a global reach you most likely need to provide the application in different languages. In order to not write the same application for each of the supported languages, this section describes how a Web application can be efficiently internationalized together with the Dojo Toolkit. It covers:

- ▶ String translation techniques
- ▶ I18n in Dijit (date, number, currency, color)
- ▶ String encoding
- ▶ Bidirectional layout of elements

## 12.5.1 Dojo i18n

This section covers Dojo's support for internationalizing strings.

### Directory structure

To internationalize a Web application with Dojo, you need a bundle and a resource for each language that you intend to support. The Dojo bundle is the directory that contains the resource, which is a JavaScript file that contains the translation of the strings for the specific language. An example of a directory structure is shown in Figure 12-16.

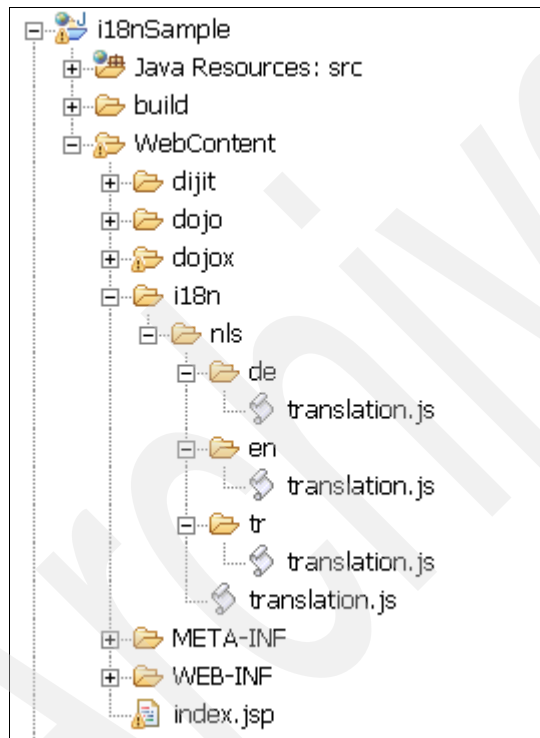


Figure 12-16 Directory structure for JavaScript resource bundles

All bundles must be placed under a root folder named `nls`. The bundle name is the standard identifier for the specific language locale. The format of the identifier is defined by ISO in the document located at:

<http://www.ietf.org/rfc/rfc3066.txt>



The locales are:

- ▶ ar (Arabic)
- ▶ cs (Czech)
- ▶ da (Danish)
- ▶ de (German)
- ▶ el (Greek)
- ▶ en (English)
- ▶ es (Spanish)
- ▶ fi (Finnish)
- ▶ fr (French)
- ▶ hu (Hungarian)
- ▶ it (Italian)
- ▶ ja (Japanese)
- ▶ ko (Korean)
- ▶ nl (Dutch)
- ▶ no (Norwegian)
- ▶ pl (Polish)
- ▶ pt (Portuguese)
- ▶ pt-br (Portuguese-Brazil)
- ▶ ru (Russian)
- ▶ tr (Turkish)
- ▶ zh (Chinese)
- ▶ zh-tw (Chinese-Taiwan)

The resources are placed under the bundle folder for each language. One default resource in English (or preferred default language) is placed at the same level as the bundles under nls. This is the resource that the application falls back to for languages that do not have a translation.

Dojo uses a JSON-object as the format of the JavaScript resource file content. Example 12-7 shows a resource file in English.

*Example 12-7 Content of English resource file*

---

```
{  
  "close" : "Close",  
  "send" : "Send"  
}
```

---

Example 12-8 shows the English resource translated into a German resource file.

*Example 12-8 Content of German resource file*

---

```
({
  "close" : "Schließen",
  "send": "Senden"
})
```

---

Dojo uses UTF-8 for the encoding of the JavaScript resource files.

## General usage

Example 12-9 shows a simple JSP to illustrate how to use Dojo's internationalization functions to access strings in resource files.

*Example 12-9 Accessing resource files with Dojo i18n*

---

```
<html>
<head>
  <script type="text/javascript">
    var djConfig = {parseOnLoad: true, locale:
      '<%= (request.getLocale().toString().replace('_',
      '-')).toLowerCase() %>'};
  </script>
  <script type="text/javascript" src="dojo/dojo.js"></script>
  <script type="text/javascript">
    dojo.require("dojo.i18n");
    dojo.require("dijit.Declaration");
    dojo.require("dojo.parser");
  </script>
  <script type="text/javascript">
    dojo.requireLocalization("i18n", "translation");
  </script>
  <script type="text/javascript">
    var bundle = dojo.i18n.getLocalization("i18n", "translation");
    dojo.addOnLoad(function() {
      dojo.byId("closeBtn").innerHTML = bundle["close"];
    });
  </script>
</head>
<body>
  <label dojoType="dijit.Declaration" widgetClass="I18NString"
    defaults="{key: ''}">
    <script type='dojo/connect' event='startup'
```

```

        this.domNode.innerHTML = bundle[this.key] ;
    </script>
</label>
<button id="closeBtn"></button>
<button id="sendBtn"><div dojoType="I18NString" key="send"></div>
</button>
</body>
</html>

```

---

In order to load the correct resource file you need to set the locale key of the `djConfig` object. You can get the locale with `request.getLocale()`. As Java uses an underscore in locale identifiers and Dojo a hyphen, you must replace all underscores.

To load a resource file `dojo.requireLocalization` is called. The first parameter is the path to the `nls` folder relative to the Dojo root folder. The second parameter is the name of the resource file, which in this example is the `translation.js` file. Dojo uses the `djConfig["locale"]` value and searches for a bundle with that name. If found, the resource file for the specific language is loaded. The default resource file is always loaded as a fallback.

To access resource strings `dojo.i18n.getLocalization` is used. It has the same parameters as `requireLocalization`. It returns a JSON object that contains at least the strings that are loaded according to the `djConfig["locale"]` value. If a value is only defined in the default resource it is also available in the resulting JSON object in the default language.

Once you have the JSON object with the resource strings you can simply access each string and use it however you wish (for example, assign it to the `innerHTML` attribute of an element).

If you do not want to populate elements with the resource strings programmatically, you can create a widget with `dijit.Declaration` and use it as shown previously in the Example 12-9 on page 582. How to use `dijit.Declaration` is described in 9.8.4, “Declaring a widget class declaratively” on page 359.

If you need to load translations for other languages in addition to the language set in the locale, set the `djConfig.extraLocale`, which expects an array of the additional languages, for example, `djConfig = {locale: "en", extraLocale: ["tr", "de"]}`. You can access these additional translations afterwards by passing a third parameter to `dojo.i18n.getLocalization`, for example, `dojo.i18n.getLocalization("i18n", "translation", "tr")`, which will return an object with the turkish strings.

## Usage in widgets

To internationalize strings in widgets it is a good practice to define placeholders for a translated string in a form like `${i18n.keyName}` (replaced with `i18n["keyName"]`, shown in the example in 9.8.7, “Advanced widgets” on page 366) in the widget HTML template and add a bundle variable that is populated from `dojo.getLocalization` to the `postMixInProperties` function of the widget. You now have a direct mapping between the placeholder and the JSON object that contains the translated strings.

## 12.5.2 Using the JSP Standard Tag Library (JSTL)

The most common file formats used for string translations are XML Localization Interchange File Format (XLIFF) (<http://docs.oasis-open.org/xliff/xliff-core/xliff-core.html>) and Gettext Portable Object (PO) (<http://www.gnu.org/software/gettext/gettext.html>). There are not many tools available for converting JSON to those formats. Therefore, if you want to use a server-side approach with JSPs for translating strings rather than a client-side with Dojo, then using the JSP Standard Tag Library (JSTL) is the best choice.

JSTL extends the JSP specification by adding JSP tags like XML processing tags, SQL tags, internationalization tags, and tags used for iteration and conditional tasks. JSTL is a way to declaratively access Java’s `i18n` API. This section only focuses on the JSTL *fmt* library that provides the internationalization tags. The complete documentation is available at:

<http://java.sun.com/products/jsp/jstl>

## Directory structure

Figure 12-17 shows how the directory structure can look when using .properties files on the server side.

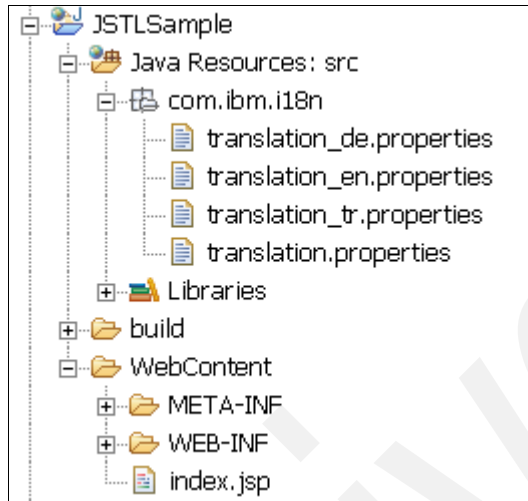


Figure 12-17 Directory structure for Java bundles for i18n

Just create a folder under Java Resources and put all your translated files into it. That folder is the bundle. The .properties file without any language extension is taken as the default resource file, in this case translation.properties. The default resource name is the basename. All other resource files must be named with the basename followed by an underscore and the locale identifier. Note that Java uses an underscore and not the hyphen that Dojo uses for the locale identifiers.

Example 12-10 shows the format of the strings in a .properties file.

Example 12-10 Content of properties file for English translation

---

```
close = Close  
send = Send
```

---

## General usage

Example 12-11 shows how you can internationalize your application with JSTL tags.

Example 12-11 Internationalization with JSTL tags

---

```
<%@ taglib uri="http://java.sun.com/jstl/fmt" prefix="fmt" %>  
<fmt:bundle basename="com.ibm.i18n.translation">  
<html>
```

```

<head>
<script type="text/javascript">
    function init() {
        document.getElementById("sendBtn").innerHTML = "<fmt:message
            key="send" />"
    }
</script>
</head>
<body onload="init()">
    <button id="closeBtn"><fmt:message key="close" /></button>
    <button title="<fmt:message key="send" />" id="sendBtn"></button>
</body>
</html>
</fmt:bundle>

```

---

First thing to do in your JSP is to define the prefix `fmt` that enables you to use the tags of the `fmt` library. With the `<fmt:bundle` `basename="com.ibm.i18n.translation">` you set up the default `.properties` file to be used for translation. The `basename` is the path to the default resource file with a dot as the separator. Note that the name of the default file without the `.properties` ending is used.

To access a string of a resource files, use the `<fmt:message key="theKey" />` statement. Use `fmt:message` to access the strings both inside a script tag (see send button in example), inside the body tag as a standalone tag (see close button), and assign it to an attribute of an HTML element (see send button title attribute).

The `fmt` library will automatically check the locale of the browser request and load the strings from the respective resource file. If a string to a specified key does not exist in the respective language file, the default resource is checked for that string. If the default resource does not contain it either, then the string that is returned from the `fmt` library has the format `???yourKey???`. This means that if you see strings in a UI starting and ending with three question marks, then there is no translation performed for that key.

The `fmt` library also provides tags for localizing dates and numbers. Those are not described in this book, as the Dojo Toolkit provides functionality for date, number, and currency handling, which is discussed in 12.5.6, “Currency, number, and date formatting” on page 590.

## Usage in widgets

In the previous section we described how you can use the `fmt:message` tag inside the JSP page. Since Dojo builds the markup for the widget after the JSP

replaced `fmt:message` tags on the page, you cannot use that tag inside the widget HTML snippet or in the widget JavaScript. Another solution is needed.

One solution is to manually create a JSON object with all translated strings. This can be done in a separate JSP and included in the main JSP. Example 12-12 shows an implementation that uses Java's `java.util.ResourceBundle` class to access strings in a resource bundle. The example shows a client-side usage. You can use the `ResourceBundle` class in the same way at the server side in your own Java classes or EJBs if you require internationalized strings at the backend.

*Example 12-12 Building a JSON object containing translated strings*

---

```
<%@page import="java.util.ResourceBundle"%>
<%@page import="java.util.Enumeration"%>
<script>
    //holds all translated strings
    var i18nStringsGlobal = {};
</script>
<% ResourceBundle myResources =
    ResourceBundle.getBundle("com.ibm.i18n.translation",
        request.getLocale());
    Enumeration<String> en = myResources.getKeys();
    String key = null;
    while(en.hasMoreElements()) {
        key = en.nextElement();
    %>
    <script>
        i18nStringsGlobal["<%= key %>"] =
            "<%= myResources.getString(key) %>";
    </script>
<% }
%>
```

---

Presuming that the JSP is called `i18nStrings.jsp` and is located in the same path as the main JSP `index.jsp` file, you can include this JSP at the top of your main JSP page using the `<%@ include file="i18nStrings.jsp" %>` statement.

After the `i18nStrings.jsp` is included the variable `i18nStringsGlobal` is accessible throughout the entire application, including the widgets JavaScript class, but not the widget template since the `${widgetVar}` (explained in 9.8.3, “Widget basics” on page 353) construct only allows access to variables that are defined within the widget class. Simply defining an `i18nStrings` variable in the widget class and assigning `i18nStringsGlobal` to it solves the problem. You can also use the global variable in the widget template. For example, `${i18nStrings.close}` is replaced by the value of the `close` key in the respective resource file.

### 12.5.3 Formatting strings with `dojo.string.substitute`

To display a string like “The file *filename* could not be found under *directoryname*” your first approach might be to define the two strings “The file” and “could not be found” and concatenate them both with the dynamic file name and directory name with the plus sign (+). This method is not correct since you do not know whether in other languages’ grammatical rules present those strings in the same order. A good practice is to save the string in the format like “The file `${0}` could not be found under `${1}`” in the `.properties` file and use Dojo’s `dojo.string.substitute` function to replace the placeholders with the respective values, as shown in Example 12-13.

*Example 12-13 Usage of `dojo.string.substitute`*

---

```
// theString = “The file ${0} could not be found under ${1}”  
dojo.string.substitute(theString, ["sample.txt", "/tmp/red"]);  
// returns the string “The file sample.txt could not be found under  
// /tmp/red
```

---

The placeholder `$(i)` is replaced with the element at the index `i` of the passed array as the second argument.

You can also use the format shown in Example 12-14.

*Example 12-14 Different usage of `dojo.string.substitute`*

---

```
// theString = “The file ${file} could not be found under ${dir}”  
dojo.string.substitute(theString, {file:"sample.txt", dir:"/tmp/red"});
```

---

### 12.5.4 I18n support for strings in Dojo widgets

The widgets provided in Dijit are already internationalized for many languages. The following sections provide an introduction on how you can add additional language support as well as use the Dojo functions for formatting dates, numbers, and currency.

### 12.5.5 Adding additional language support to Dijit

The Dojo Toolkit (without DojoX) provided with the Ajax client runtime contains five `nls` folders in total. The supported languages are:

- ▶ `cs` (Czech)
- ▶ `de` (German)
- ▶ `en` (English, USA, Canada, United Kingdom)



- ▶ es (Spanish)
- ▶ fr (French)
- ▶ hu (Hungarian)
- ▶ it (Italian)
- ▶ ja (Japanese)
- ▶ ko (Korean)
- ▶ pl (Polish)
- ▶ pt (Portuguese)
- ▶ ru (Russian)
- ▶ zh (Chinese)
- ▶ zh-tw (Chinese-Taiwan)

The directories containing translations for the respective widgets that need to be internationalized are:

- ▶ Editor widget translations: Tooltips for the Editor buttons like “Copy”, “Paste”, or “Cut”. Located under `<dojo-root>/dijit/_editor/nls`.
- ▶ ComboBox, TextArea translations, and validation messages: Located under `<dojo-root>/dijit/form/nls`.
- ▶ Common string translations used by several widgets: Strings like “Ok”, “Cancel”, and “Loading”. Located under `<dojo-root>/dijit/nls`.
- ▶ Translations for colors: Used as a tooltip for color names when you hover over colors on the ColorPalette widget. Located under `<dojo-root>/dojo/nls`.
- ▶ Currency, date, and number: Used for formatting locale-specific currencies, dates, and numbers. Located under `<dojo-root>/dojo/cldr/nls`.

As explained in 12.5.1, “Dojo i18n” on page 580, each locale has its own bundle folder. If you want to support additional languages, just copy one of the existing bundles with the JavaScript resources to it, rename that bundle with the locale of the country in which you want to support the application, and replace the string values of each key in the .js resource file with the respective translation.

The JavaScript resources in the first four directories of the list contain key-value mappings in the format:

key: “valueAsString”

Those strings are Dojo specific. The resource files used for formatting currencies, dates, and numbers are derived from the Common Locale Data Repository (CLDR). The CLDR is a large standard repository of locale data. The Web site is:

<http://unicode.org/cldr>

All locale files used to format currencies, dates, and numbers are provided in Locale Data Format Language (LDML) format, an XML-based markup language.

To use the locale data in Dojo you must convert the locale file from CLDR into the proper JavaScript resource file. Dojo provides an ANT-script that converts specified LDML files into JSON format and adds them to your `<dojo-root>/dojo/cldr` directory automatically. The CLDR-tool is not part of the Ajax client runtime. It can be accessed at:

<http://archive.dojotoolkit.org/nightly/checkout/util/buildscripts/cldr>

This Web site contains a description of the usage and also all current LDML files of the CLDR for many languages.

## 12.5.6 Currency, number, and date formatting

This section describes how to use Dojo's currency, date, and number formatting functions.

### Number and currency

Example 12-15 shows how to use the `dojo.number.format` function to format numbers according to different locales. Example 12-16 shows the usage of the `dojo.currency.format` function.

*Example 12-15 Formatting numbers with dojo.number*

---

```
dojo.require("dojo.number")
...
1. dojo.number.format(1234567)
2. dojo.number.format(123.4567))
```

results with locale = "en-us":

1. 1,234,567
2. 123.457

results with locale = "de":

1. 1.234.567
2. 123,457

---

Dojo automatically formats the number according to the conventions (decimal and thousands separator, symbols used to indicate exponential numbers and percentages) for the locale that you assign to the `djConfig` variable.

*Example 12-16 Formatting currencies with dojo.currency*

---

```
dojo.require("dojo.currency")
...
1. dojo.currency.format(1234567, {currency: "EUR"})
```

```
2. dojo.currency.format(123.4567, {currency: "USD"})
3. dojo.currency.format(123.4567, {currency: "JPY"})
```

results with locale = "en-us":

```
1. ?1,234,567.00
2. $123.46 //Dojo automatically rounds the currency
3. ¥123
```

results with locale = "de":

```
1. ? 1.234,567,00
2. $ 123,46
3. ¥ 123
```

---

The currency value that is passed to the function is a three-letter symbol defined by the ISO. A list of all symbols is available at:

[http://en.wikipedia.org/wiki/ISO\\_4217](http://en.wikipedia.org/wiki/ISO_4217)

## Date

Example 12-17 shows how to use `dojo.date.locale.format` to format JavaScript *date* objects.

*Example 12-17 Formatting dates with dojo.date.locale*

---

```
dojo.require("dojo.date.locale")
...
var d = new Date(2008, 4, 30, 22, 10, 34)
1. dojo.date.locale.format(d)
2. dojo.date.locale.format(d,{formatLength: "medium"})
3. dojo.date.locale.format(d,{selector: "date", formatLength: "full"})
4. dojo.date.locale.format(d,{selector: "time", formatLength: "short"})
```

results with locale = "en-us":

```
1. 5/30/08 10:10 PM
2. May 30, 2008 10:10:34 PM
3. Friday, May 30, 2008
4. 10:10 PM
```

results with locale = "de":

```
1. 30.05.08 22:10
2. 30.05.2008 22:10:34
3. Freitag, 30. Mai 2008
4. 22:10
```

---

## 12.5.7 String encoding

To ensure that all characters for each language supported by your Web application are displayed correctly you should use UTF-8 encoding since the default ISO-8859-1 encoding does not handle all Unicode characters. A detailed description of both Unicode and UTF-8 encoding can be found at:

<http://www.utf-8.com>

## 12.5.8 UTF-8 encoding for the client side

Example 12-18 shows how UTF-8 encoding can be applied to a simple HTML page.

*Example 12-18 Setting UTF-8 encoding in an HTML document*

---

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
</body>
</html>
```

---

The attribute `charset` in the meta tag defines the character encoding used to send the content from the server to the browser. For XML pages you should define the tag `<?xml version="1.0" encoding="UTF-8"?>` at the top of your page since the META tag cannot be used. Example 12-19 shows how to set up UTF-8 encoding in a JSP.

*Example 12-19 Setting UTF-8 encoding in a JSP*

---

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
</body>
</html>
```

---

The `pageEncoding` attribute tells the JSP compiler how to read the bytes in the JSP file. The `contentType` attribute tells the compiler which encoding to use and it also sets the content-type of the header to UTF-8.

## 12.5.9 UTF-8 encoding on the server side

If you do not specify an encoding type for the request on the server side (that is, in your JSP or servlet) the server uses its default encoding type to decode the parameters in the request. Most servers do not use UTF-8 as the default. For instance, the WebSphere Application Server 6.1 uses a locale map that lists the encoding type to be used according to the request's locale.

If you want to use UTF-8 on the server side, simply set the request encoding explicitly, as shown in Example 12-20 for a JSP.

*Example 12-20 Setting UTF-8 encoding of a client request*

---

```
<%@page contentType="text/html; charset=utf-8" pageEncoding="utf-8"%>
<% request.setCharacterEncoding("utf-8"); %>
...
<% String param = request.getParameter("param"); %>
```

---

An example of setting UTF-8 encoding explicitly for a servlet on the server side is shown in Example 12-21.

*Example 12-21 Setting UTF-8 encoding of a response in a servlet*

---

```
protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
    request.setCharacterEncoding("UTF-8");
    response.setContentType("text/html; charset=UTF-8");
    //...
    String param = request.getParameter("param");
    //...
}
```

---

Set the encoding type of the response to UTF-8 if you want to ensure that the response is also encoded UTF-8.

## 12.5.10 Dojo and UTF-8 encoding of request parameters

The encoding of the HTML or JSP file works fine if you load the complete page. In an Ajax-application this is normally done only once when you first start the application in the browser. What do you do when sending an Ajaxarequest to the server that does not cause the entire page to reload? You must encode the parameters that are sent to the server and decode the responses that are returned.

JavaScript provides the functions `encodeURIComponent(String value)` and `decodeURIComponent(String value)` to encode a string with UTF-8 and decode a UTF-8 encoded string.

If you use the Dojo `xhr*`-functions you need not worry about encoding. Dojo automatically encodes the parameters and decodes the response using the `encodeURIComponent` function. This encoding/decoding is only applied as long as the parameters are not attached to the URL of the request but supplied as the content property, as shown in Example 12-22.

*Example 12-22 UTF-8 encoding with `encodeURIComponent`*

---

```
dojo.xhrGet ({
  url: "/i18nDojo/Test?param1=" + encodeURIComponent("\u00a9"),
  content: { param2: "\u00a9" },
  load: function (data) { /*...*/ },
  error: function (data) { /*...*/ }
});
```

---

If you pass the parameters as the content argument to the `xhr` function, Dojo does the encoding. If you pass it to the URL directly, you have to do the encoding. Otherwise, the string may be uninterpretable on the server side.

### 12.5.11 Bidirectional (BiDi) layout of elements

In some languages like Arabic or Hebrew the page flow is from right to left (RTL). This means that characters in a sentence and UI elements are in reverse order compared to a user interface in the English language.

## BiDi support in browsers

Browsers have already integrated some RTL support. Since the default page layout is left-to-right, in order to set the browser into RTL mode you need to set the `dir` attribute of the HTML or BODY tag to `rtl`. See Figure 12-18.

```
<html dir="rtl">
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
</head>
<body>
  <div dir="ltr">
    <div>english1 english2 english3</div>
    <div>english1 ايتخال english2 الففن english3</div>
    <div>english1 ايتخال english2 english3</div>
    <div>english1 english2 english3 ايتخال</div>
    <div>ايتخال english1 english2 english3</div>
  </div>
  <div>
    <div>english1 english2 english3</div>
    <div>english1 ايتخال english2 الففن english3</div>
    <div>english1 ايتخال english2 english3</div>
    <div>english1 english2 english3 ايتخال</div>
    <div>ايتخال english1 english2 english3</div>
  </div>
  <br/>
  <table>
    <tr><td>Column1</td><td>Column2</td><td>Column3</td></tr>
  </table>
  <table dir="ltr">
    <tr><td>Column1</td><td>Column2</td><td>Column3</td></tr>
  </table>
  <div dir="ltr">
    <table dir="rtl">
      <tr><td>Column1</td><td>Column2</td><td>Column3</td></tr>
    </table>
  </div>
  <div dir="ltr">
    <table>
      <tr><td>Column1</td><td>Column2</td><td>Column3</td></tr>
    </table>
  </div>
</body>
</html>
```

Figure 12-18 Usage of the HTML `dir` attribute

If you set the dir attribute to RTL in the HTML tag, then the main page flow goes from right to left. This means that if you do not override that attribute's value with LTR in an element in the body, the element's dir-attribute is RTL since it is inherited from the HTML element.

RTL mode automatically aligns elements to the right side of the window. The scroll bar appears on the left side. You can combine RTL and LTR alignment by overriding the dir attribute.

In RTL mode, if you have a sentence that only contains non-Arabic and non-Hebrew words (LTR-words), the sentence will be aligned to the right, but the words in the sentence are in the same order in which you define them in the HTML markup. The browser groups LTR and Arabic/Hebrew words and displays the groups in reverse order. For instance, if you have “english1 arabic1 english2 english3“, the grouping is (english1) (arabic1) (english2 english3) and the final result displayed is “(english2 english3) (arabic1) (english1)“.

Setting the dir attribute to RTL in a table element causes the table columns to be displayed in reverse order. That is handled automatically by the browser.

Figure 12-19 shows an example of this output displayed using the Internet Explorer 6 browser.

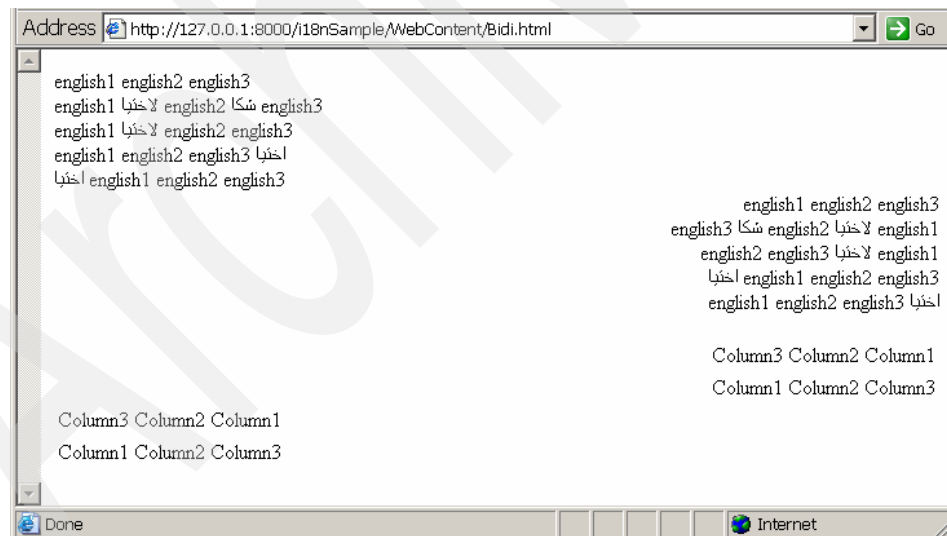


Figure 12-19 HTML page that uses RTL page direction



If you want the JSP to dynamically set the page direction according to the language, you can use the snippet shown in Example 12-23.

*Example 12-23 Setting the HTML dir attribute dynamically*

---

```
<% String lang = request.getLocale().getLanguage();
    //ar = arabic languages, iw = hebrew language
    String dir = (lang.equalsIgnoreCase("ar") ||
        lang.equalsIgnoreCase("iw")) ? "rtl" : "ltr";
%>
<html dir="<%= dir %>">
```

---

Using tables for laying out elements is convenient. However, it is not always a good practice because the use of tables may reduce code readability. Using div-elements has more flexibility in positioning of elements. Since a div-element causes a line break, you have to use the CSS attribute float to display div-elements in the same line.

Example 12-24 shows how you should style div-elements in LTR and RTL mode.

*Example 12-24 Mix of LTR and RTL mode*

---

```
<html>
<head>
<style>
    .divLtr {
        float: left;
    }
    .divRtl {
        float: right;
    }
</style>
</head>
<body>
    <div class="divLtr">1</div>
    <div class="divLtr">2</div>
    <div class="divLtr">3</div>

    <div class="divRtl">1</div>
    <div class="divRtl">2</div>
    <div class="divRtl">3</div>
</body>
</html>
```

---

This example instructs the browser to display the numbers 123 in the top left corner of the window and the numbers 321 in the top right corner of the window.

## BiDi support in Dojo

Widgets in Dojo 1.1, except for deprecated widgets, are BiDi capable. However, the extent of the DojoX BiDi support varies. The following widgets have limited BiDi support in Version 1.0 for the tundra theme:

```
dijit.ColorPalette, dijit.Declaration, dijit.Dialog, dijit.Menu,  
dijit.ProgressBar, dijit.TitlePane, dijit.ToolBar, dijit.Tooltip,  
dijit.Tree, dijit.form.Button, dijit.form.CheckBox,  
dijit.form.ComboBox, dijit.form.CurrencyTextBox,  
dijit.form.DateTextBox, dijit.form.FilteringSelect, dijit.form.Form,  
dijit.form.InlineEditBox, dijit.form.NumberSpinner,  
dijit.form.NumberTextBox, dijit.form.Textarea, dijit.form.TextBox,  
dijit.form.TimeTextBox, dijit.form.ValidationTextBox,  
dijit.layout.ContentPane, dijit.layout.LinkPane,  
dijit.layout.StackContainer, dijit.layout.AccordionContainer
```

To enable RTL styling for widgets you must include the separate CSS file `tundra_rtl.css` (or `soria_rtl.css`) within your page following the `tundra.css`. The RTL CSS file is in the same folder as the `tundra.css` (or `soria.css`, respectively).

Dojo automatically adds the class `.dijitRtl` to the body if the `dir` attribute of the page is set to RTL. This means that `.dijitRtl` is inherited by all elements in the body and thus CSS can be used to apply different styling to elements in RTL mode.

Example 12-25 shows tundra styles for the `.dijitTab` (applied to tabs in a `TabContainer` widget) class.

### *Example 12-25 CSS definition with .dijitRtl usage*

---

Content in `tundra.css`:

```
.tundra .dijitTab {  
    line-height:normal;  
    margin-right:5px;  
    padding:0px;  
    border:1px solid #ccc;  
    background:#e2e2e2 url("images/tabEnabled.png") repeat-x;  
}
```

Content in `tundra_rtl.css`:

```
.dijitRtl .dijitTab {  
    margin-right:0;  
    margin-left:5px;  
}
```

---

When including first the `tundra.css` and then the `tundra_rtl.css`, the `margin-right` style is overridden by the one defined for `.dijitRtl`. The resulting styles applied to `.dijitTab` in RTL mode for the snippet above are shown in Example 12-26.

*Example 12-26 CSS definitions applied to a tab in RTL mode*

---

```
line-height:normal;
margin-right:0;
margin-left:5px;
padding:0px;
border:1px solid #ccc;
background:#e2e2e2 url("images/tabEnabled.png") repeat-x;
```

---

In a well-designed Web application the RTL styles should only contain styles for the layout and positioning of elements that override the common styles, including font sizes and so on. This means that in the best case you should only have to override styles like `float`, `padding`, `border`, `margin`, `position`, `text-align`, and so on, when using the `.dijitRtl` in your own CSS files.

## 12.6 Accessibility

In order to allow visually impaired people to use your application or to use the application without a mouse or other devices like a keyboard you must spend additional effort on development. This section discusses the areas that you must consider in order to make your application accessible to different user groups. It covers:

- ▶ Device-independent interaction
- ▶ Sizing issues
- ▶ High-contrast mode and turning off images
- ▶ Assistive technology support
- ▶ Accessibility support in available Dojo widgets

### 12.6.1 Device-independent interaction

Since not all users are capable of using a mouse, keyboard support for your application should be provided. HTML links and form controls can be automatically accessed with the Tabulator key.

#### Setting focus to an element

Since widgets use both links and form controls, you must implement the respective key handling manually. To enable focusing of an element the `tabindex` attribute can be used.

If the `tabindex` attribute is not present then the element can only be focused with the Tabulator key if it is a form control or an anchor tag. If `tabindex=0`, then the element can be focused in tab order relative to its position in the document. If `tabindex` is a positive integer between 1 and 32768 then the number specifies this element's position in the tab order. If `tabindex=-1` the element is not in the tab order and cannot be focused with the Tabulator key. It must be focused manually using the `element.focus()` statement.

Accessing each element on the page using only the Tabulator key can be tedious. For a more intuitive navigation on the page, it is best to enable focusing with additional keys such as the arrow keys. The `tabindex` must be -1 in order to focus an element as the result of pressing an arrow key. You can define a `keydown` event handler in order to determine the next element to receive the focus. For instance, Dojo implements this technique in its `TabContainer` widget. Once one of the Tabs has a focus you can switch between the previous or next Tab by pressing the left or right arrow key.

As a general recommendation, use the Tabulator key to navigate between components and arrow keys or other keys to navigate inside components. Do not use CSS to simulate focusing an element, but use `element.focus()` instead, as window readers do not read element names when the style is changed.

## 12.6.2 Sizing issues

As some users prefer larger or smaller font sizes, browsers support increasing or decreasing the size of fonts displayed on a Web page.

As IE6 and IE7 do not allow magnifying or demagnifying text specified in an absolute size, do not use pixel sizing of fonts if you want to offer this capability in your application.

Try to avoid pixel sizes for elements containing text, as magnifying the text could exceed the borders of the container. For instance, if you have a `div` tag with a height of 16px and text inside the `div` with a font size of 1em, the text will cut the `div`'s border once you magnify the text in the browser, for example, `Ctrl + '+'` for the Firefox browser. If you set the `div` height to 1em (or greater) the height of the `div` grows dynamically as well and no problems occur.

## 12.6.3 High-contrast mode and turning off images

You can use images via the `img` tag in your HTML code (or dynamically in JavaScript) or by setting background properties in your CSS file. If you use CSS background images you can easily customize the look and feel of your application by simply overriding the background property. Using CSS compared

to the `img` tag also has the advantage that all images in one CSS file are loaded by a single request, whereas each `img` tag needs a separate request to the server. If you use image tags, it is good practice to set the `alt` attribute to show a text alternative in case the image cannot be displayed.

Using CSS background images has the drawback that images are not displayed when the browser is in high contrast mode. To check whether high contrast mode is enabled, you can create a `div` tag, set the color of two borders to a different value, obtain the color value afterwards, and compare both values. If they are the same the page is displayed in high contrast mode, since that mode overrides the CSS colors with the high contrast theme color. To check whether images are turned off, create a `div` element, set the background image, and check using the JavaScript function `getComputedStyle` whether the returned value is `none` (for Internet Explorer), `null`, or `url(invalid-url:)` (for Firefox). If yes, then images are disabled for the browser running the application.

If the application is currently running in high contrast mode or images are turned off Dojo automatically sets an additional class `dijit_a11y` to the body of the HTML document. You can use that class in your CSS file to define specific styling in case images are not displayed. To make this clearer, let us look at how the HTML code for the close icon of the Dialog widget looks. This is shown in Example 12-27.

*Example 12-27 HTML template snippet of the Dialog widget*

---

```
<span dojoAttachPoint="closeButtonNode" class="dijitDialogCloseIcon"
dojoAttachEvent="onclick: onCancel">
  <span dojoAttachPoint="closeText" class="closeText">x</span>
</span>
```

---

Example 12-28 shows parts of the CSS definitions for `dijitDialogCloseIcon` and `closeText` that define how these elements are displayed.

*Example 12-28 CSS definitions for a dialog widget in high contrast mode*

---

```
.tundra .dijitDialogCloseIcon {
  background : url("images/spriteRoundedIconsSmall.png")
}

.dijitDialog .closeText {
  display:none;
}

.dijit_a11y .dijitDialog .closeText {
  display:inline;
}
```

---

As we see in normal display mode, the span with the class closeText is not displayed. In high contrast mode it is made visible, and the alternate text x is used as the close icon since the original image is not displayed.

Figure 12-20 shows a dialog widget in the high contrast color mode white.

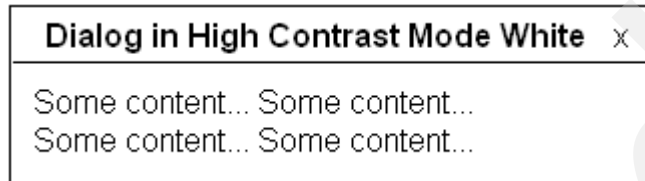


Figure 12-20 Dialog widget in high contrast mode with white theme color

## 12.6.4 Assistive technology support

Some HTML elements like lists, forms, and links have defined roles and states like checked, unchecked, disabled, visited, and so on, that are accessed by assistive technologies like a screen reader. Since widgets use elements like span or div we need to define roles and states for those elements in order to provide detailed information about a user interface component to the assistive technology.

The W3C Web Accessibility Initiative (WAI) provides an Accessible Rich Internet applications (ARIA) specification that can be used to define roles and states for HTML elements.

### ARIA specification

For detailed information about all ARIA documents see:

<http://www.w3.org/WAI/intro/aria.php>

All supported WAI roles like tree, treeitem, grid, gridcell, and so on, can be found at:

<http://www.w3.org/TR/2007/WD-aria-role-20071019>

Available WAI states can be seen at:

<http://www.w3.org/TR/2007/WD-aria-state-20071019>

Figure 12-21 shows how ARIA terms are applied to a tree widget.

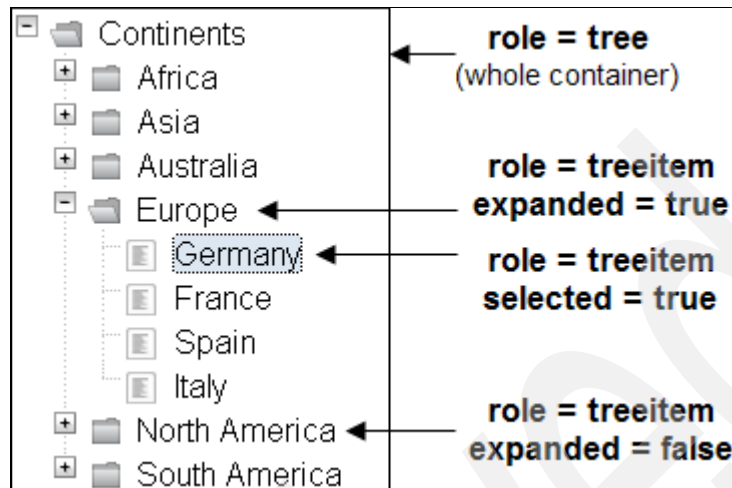


Figure 12-21 Roles and states of a tree widget

ARIA support is implemented in Firefox 1.5+ and will be available for Internet Explorer 8. Opera has begun to add support to the Opera browser (9.5 and later). Webkit, the open source framework behind Safari, announced its intent to add support for future releases. Screen reader support is available in Windows-Eyes 5.5+ and JAWS 7.1+.

## Dojo and ARIA

The ARIA roles and states are attributes that are added to the element's HTML portion. Dojo implemented the ARIA specification. Roles and states for all widgets are set, respectively.

Widget roles and states that are known when a widget is instantiated are directly added to the widget's HTML template. Dynamic change of roles and states can be done with following methods that Dojo provides.

- ▶ WAI roles:
  - `dijit.hasWaiRole(/*HTML element*/ el)`: Checks whether a role is set for the element.
  - `dijit.getWaiRole(/*HTML element*/ el)`: Obtains the role set. It is an empty string if no role is set.
  - `dijit.setWaiRole(/*HTML element*/ el, /*String*/ role)`: Sets a role for the element, for example, `dijit.setWaiRole(someDiv, "treeitem")`.
  - `dijit.removeWaiRole(/*HTML element*/ el)`: Removes the role attribute from the element.

- ▶ WAI states:
  - `dijit.hasWaiState(/*HTML element*/ el, /*String*/ state)`: Checks whether the state is set for the element.
  - `dijit.getWaiState(/*HTML element*/ el, /*String*/ state)`: Gets the value of the state for the element. It is empty if the state is not set.
  - `dijit.setWaiState(/*HTML element*/ el, /*String*/ state, /*String*/ value)`: Sets a state for the element, for example, `dijit.setWaiState(someDiv, "disabled", "true")`.
  - `dijit.removeWaiState(/*HTML element*/ el, /*String*/ state)`: Removes the specified state from the element.

### 12.6.5 Accessibility support in available Dojo widgets

All widgets in the Dijit package are fully accessible using the keyboard. High contrast mode is supported in Firefox 2.0+ and Internet Explorer 6+. Support for turning off images is provided for Firefox. The ARIA specification is implemented for all widgets.



## Rich Internet applications

This chapter outlines approaches that can be used by other Rich Internet application frameworks to access services created using the WebSphere Application Server Feature Pack for Web 2.0.

This chapter provides examples of how these technologies can be used in conjunction with the features of the Web 2.0 FEP.

It also discusses other platforms that can be used to integrate with Web 2.0 services built using the Web 2.0 Feature Pack.

This chapter provides a detailed example of using the Web 2.0 Feature Pack functions with the Adobe® AIR and Flex 3 products.

## 13.1 Rich Internet applications

Rich Internet applications (RIAs) are Web applications that have the features and functionality of traditional desktop applications. RIAs typically transfer the processing necessary for the user interface to the Web client but keep the bulk of the data on the application server. They can also adapt to network connectivity and store data locally if needed. They also leverage the latest UI technologies to present a compelling, easy-to-use interface.

RIAs typically:

- ▶ Run in a Web browser, or do not require software installation
- ▶ Run locally in a secure environment called a sandbox

Traditional Web applications are based on a client-server architecture utilizing a thin client. In this design most all processing is performed on the server, and the client is only used to display static (in this case HTML) content. The biggest drawback with this approach is that all interactions with the application must involve the server, which requires data to be sent to the server, the server to respond, and the page containing the response to be reloaded on the client.

Internet standards have evolved slowly and continually over time to accommodate these techniques, so it is hard to draw a strict line between what constitutes an RIA and what does not. But all RIAs share one characteristic: They introduce an intermediate layer of code, often called a client engine, between the user and the server. This client engine is usually downloaded at the start of the application, and may be supplemented by further code downloads as the application progresses. The client engine acts as an extension of the browser, and usually takes responsibility for rendering the application's user interface and for handling server communication.

However, what can be accomplished in an RIA may be limited by the capabilities of the client platform. In general, the client engine is designed to perform application functions that enhance some aspect of the user interface or improve its responsiveness when handling certain user interactions, compared to the traditional Web application implementation. While adding a client engine does not require an application to depart from the traditional synchronous pattern of interactions between browser and server, in most RIAs the client engine performs additional asynchronous communications with servers.

### 13.1.1 Benefits of an RIA

Although developing applications to run in a Web browser is a more difficult process than developing a traditional desktop application, the extra effort is usually justified because:

- ▶ Local installation is not required. The application resides on the server.
- ▶ Updates/upgrades to new versions are automatic.
- ▶ Users can access the application from any computer with an Internet connection and, in most cases, regardless of what operating system that computer is running.

Because RIAs employ a client engine to interact with the user they provide the following advantages:

- ▶ Move some elements of the Model View Controller design pattern from the server to the client run time. This enables a far greater level of control for the user interaction and the data that is requested from the server. It allows for intelligent decisions to be made about when and how much data to retrieve, what to cache, and when to make a decision to get more. This pattern is very common in RIAs that use a wide variety of data volumes.
- ▶ Richer. They can offer user-interface behaviors not normally available using only the HTML widgets available to standard browser-based Web applications. This richer functionality may include anything that can be implemented in the technology being used on the client side, including drag and drop, using a slider to change data, and calculations performed only by the client and that do not need to be sent back to the server.
- ▶ Client/server balance. The demand for client and server computing resources is better balanced, so the Web Application Server need not be the workhorse that it is with a traditional Web application.
- ▶ Asynchronous communication. The client engine can interact with the server without waiting for the user to perform an action such as selecting a button or link. This allows the user to view and interact with the page independent of the client engine's communication with the server. This option allows RIA designers to move data between the client and the server without making the user wait. Perhaps the most common use of this capability is prefetching, in which an application anticipates a future need for certain data and downloads it to the client before the user requires it, thereby speeding up a subsequent response. Google Maps uses this technique to move adjacent map segments to the client in anticipation of the user scrolling them into view.
- ▶ Network efficiency. The network traffic may also be significantly reduced because an application-specific client engine can be more intelligent than a standard Web browser when deciding what data needs to be exchanged with

servers. This can speed up individual requests or responses because less data is being transferred for each interaction, and overall network load is reduced. However, use of asynchronous prefetching techniques can neutralize or even reverse this potential benefit. Because the code cannot anticipate exactly what every user will do next, it is common for such techniques to download extra data, not all of which will be used.

### 13.1.2 Limitations

With all the excitement that is being channeled towards RIA solutions it very important to realize that they have some shortcomings and cannot solve every application need. Limitations and restrictions associated with RIAs are:

- ▶ Sandboxing. Because RIAs run within a sandbox, they have restricted access to system resources. If assumptions about access to resources are incorrect, RIAs may fail to operate correctly. Adobe and Microsoft deliver a sandbox as a browser plug-in.
- ▶ Enabling scripting. JavaScript or another scripting language is often required. If the user has disabled active scripting in her browser, the RIA may not function properly, if at all.
- ▶ Client processing speed. To achieve platform independence, some RIAs use client-side scripts written in interpreted languages such as JavaScript, with a consequential loss of performance (a serious issue with mobile devices). This is not an issue with compiled client languages such as Java, where performance is comparable to that of traditional compiled languages, or with Adobe Flash® or Microsoft Silverlight™, in which the compiled code is executed by a Flash or Silverlight plug-in.
- ▶ Script download time. Although it does not have to be installed, the additional client-side intelligence (or client engine) of RIA applications needs to be delivered by the server to the client. While much of this is usually cached it still needs to be transferred at least once. Depending on the size and type of delivery, script download time may be undesirably long. RIA developers can lessen the impact of this delay by compressing the scripts and by staging their delivery to occur over multiple page loads of the application.
- ▶ Loss of integrity. If the application base is X/HTML, conflicts arise between the goal of an application (which naturally wants to be in control of its presentation and behavior) and the goals of X/HTML (which naturally wants to give away control). The DOM interface for X/HTML makes it possible to create RIAs, but by doing so makes it impossible to guarantee correct function. Because an RIA client can modify the RIA's basic structure and override presentation and behavior, it can cause failure of the application to work properly on the client side. Eventually, this problem could be solved by new client-side mechanisms that granted an RIA client more limited permission to

modify only those resources within the scope of its application. (Standard software running natively does not have this problem because by definition a program automatically possesses all rights to all its allocated resources.)

- ▶ Loss of visibility to search engines. Search engines may not always be able to index the text content of an RIA application.
- ▶ Dependency on an Internet connection. An ideal network-enabled replacement for a desktop application would allow users to be *occasionally connected*, wandering in and out of hot-spots or from office to office. This architecture is being realized by Google Gears, Lotus Expeditor, and Adobe AIR platforms.
- ▶ Accessibility. There are a lot of known Web accessibility issues in RIA, most notably the fact that screen readers have difficulty detecting dynamic changes (caused by JavaScript) in HTML content.

**Note:** The advent of RIA technologies has introduced considerable additional complexity into Web applications. Traditional Web applications built using only standard HTML, having a relatively simple software architecture and being constructed using a limited set of development options, are relatively easy to design and manage. For the person or organization using RIA technologies to deliver a Web application, their additional complexity makes them harder to design, test, measure, and support.

- ▶ RIA architecture does not follow the classic Web page paradigm. Traditional Web applications can be viewed as a series of Web pages, each of which requires a distinct download, initiated by an HTTP GET request. This model has been characterized as the Web page paradigm. RIAs invalidate this model, introducing additional asynchronous server communications to support a more responsive user interface. In RIAs, the time to complete a page download may no longer correspond to something a user perceives as important, because (for example) the client engine may be prefetching some of the downloaded content for future use. New measurement techniques must be devised for RIAs to permit reporting of response time quantities that reflect the user's experience. In the absence of standard tools that do this, RIA developers must instrument their application code to produce the measurement data needed for SLM.

### 13.1.3 RIA run times

Rich Internet applications are still in the early stages of development and user adoption. A number of restrictions and requirements that remain are:

- ▶ **Browser adoption:** Many RIAs require modern Web browsers in order to run. Advanced JavaScript engines must be present in the browser in support of techniques such as XMLHttpRequest for client-server communication, DOM scripting, and advanced CSS techniques to enable the rich user interface.
- ▶ **Web standards:** Differences between Web browsers can make it difficult to write an RIA that will run across all major browsers. The consistency of the Java platform, particularly after Java 1.1, makes this task much simpler for RIAs written as Java applets.
- ▶ **Development tools:** Some Ajax frameworks and products such as Curl, Adobe Flex™, and Microsoft Silverlight provide an integrated environment in which to build RIAs.
- ▶ **Accessibility concerns:** Additional interactivity may require technical approaches that limit applications' accessibility.
- ▶ **User adoption:** Users expecting standard Web applications may find that some accepted browser functionality (such as the Back button) may have somewhat different or even undesired behavior.

#### JavaScript/Ajax

The first major client-side language and technology available for executing code and incorporated into a majority of Web browsers was JavaScript. Although its functions were relatively limited at first, combined with layers and other developments in DHTML, it has become possible to piece together an RIA system without the use of a unified client-side solution. Ajax is a new term coined to refer to this combination of techniques and has recently been used most prominently by Google for projects such as Gmail and Google Maps. However, creating a large application in this framework can be difficult, as many different technologies must interact to enable it to work, and cross-browser incompatibilities may require a significant effort. In order to make the process easier, several open source Ajax frameworks have been developed.

#### Google's GWT framework

Google released the Google Web Toolkit (GWT), which supports the development and testing of JavaScript-based Ajax RIA's using the Java language. The GWT programming paradigm centers around coding user interface logic in Java (similar to the swing/AWT model) and then using the GWT compiler to translate this logic into cross-browser-compatible JavaScript. Designed specifically for Java developers, GWT enables Java programming, refactoring, debugging, and unit testing of RIAs using existing tools (for example,

Eclipse), without requiring knowledge of JavaScript or specific browser DOM irregularities, although hand-written JavaScript can still be used with GWT if desired.

## **Adobe Flash, Adobe Flex, and Adobe AIR**

Adobe Flash is another way to build Rich Internet applications. This technology is cross-platform and quite powerful to create an application UI.

Adobe Flex is a framework that provides the developer with the ability to create user interfaces by compiling MXML™, an XML-based interface description language. This Adobe Flex framework is compiled and turned into a SWF file to be run in the Adobe Flash player.

Adobe has also released Adobe Integrated Runtime (AIR), which is a runtime platform that is independent of the hosting operating system. Adobe AIR allows for Flash Player and Ajax applications to be deployed and installed onto a user's desktop much as you would a desktop application.

## **OpenLaszlo**

OpenLaszlo is an open source rich Internet application framework developed by Laszlo Systems Inc. The OpenLaszlo server compiles programs written in the LZX language (a mixture of XML tags and JavaScript) into either DHTML or Adobe Flash bytecode. The server was open sourced under the Common Public License. OpenLaszlo is the only rich Internet application platform that is capable of compiling into two different run times from the same code base.

## **Curl**

Curl began as a research project at MIT in the late 1990s. It was commercialized by Curl Inc, which made a first release in 2002. The current release is Version 6.0, available for Windows, Linux, and Mac clients. There is no required server-side component. Any Web server can be used. Curl provides a wealth of features, but in an easily accessible form that allows users with a variety of backgrounds to work at different levels of complexity, from simple HTML-like formatting to sophisticated object-oriented programming. One advantage of the plug-in architecture is that Curl applets work the same on every platform, with any browser, with the exception of a few platform-specific APIs. Curl applets are compiled to machine code for fast execution, and various caching mechanisms speed up the loading of applets. Curl is free to use for non-commercial and some commercial uses (see licensing). A Pro version is available, which provides additional enterprise class capabilities.

Curl has had a feature of *detached applets* for several years, which is a Web deployed applet that can run independently of a browser window, similarly to Adobe AIR. Curl applets can also be written so that they will run when

disconnected from the network. In fact, the Curl IDE is an application written in Curl.

## **JavaFX**

Sun Microsystems has announced JavaFX™, a family of products based on Java technology designed to provide a consistent experience across a wide variety of devices including desktops as applets or stand-alone clients, set-top boxes, mobile devices, and Blu-Ray players. The JavaFX platform will initially comprise JavaFX Script and JavaFX Mobile. Invented by Sun Software Engineer Chris Oliver as a skunk works project, JavaFX Script enables rapid development of rich 2D interfaces using a declarative syntax similar to SVG. Sun plans to release JavaFX Script as an open source project, but JavaFX Mobile will be a commercial product available through an OEM license to carriers and handset manufacturers.

## **Java applets**

Java applets run in standard HTML pages and generally start automatically when their Web page is opened with a modern Web browser. Java applets have access to the window (inside an area designated in its page's HTML) and the speakers, keyboard, and mouse of any computer on which their Web page is opened, as well as access to the Internet, and provide a sophisticated environment capable of real-time applications.

## **Java applications**

Java-based RIAs can be launched from within the browser or as free-standing applications via Java Web Start, which integrate with the desktop. Java RIAs can take advantage of the full power of the Java platform to deliver rich functionality, 2D and 3D graphics, and off-line capabilities.

## **Microsoft Silverlight**

Microsoft Silverlight, which can be considered a subset of Windows Presentation Foundation (WPF) allows developers to develop RIA. Like Windows Presentation Foundation, Silverlight uses XAML. Therefore, developers with previous development experiences using the .NET Framework 3.0 and XAML will find Silverlight familiar, appealing, and easy to use. See:

<http://silverlight.net>

Client machines need to install a small (about 2 MB) plug-in (Silverlight Runtime) in order to be able to play Silverlight contents. At this time, Silverlight client for Windows and Mac OS X is available from Microsoft. A third-party open-source plug-in called Moonlight is also available for Linux. Microsoft has also promised to broaden the range of supported clients.



## 13.2 Integrating RIAs with WebSphere Application Server

As Rich Internet applications are deployed the need to integrate business data into these application will grow. The service-oriented architectures that are now underlying most business data will enable access to a wide variety of feeds and business logic. The WebSphere Application Server is perfectly positioned to support these next-generation applications by leveraging the Web 2.0 Feature Pack programming model.

This section looks at some of the most popular Rich Internet application development frameworks and give examples of how they might integrate with WebSphere. The acceptance of easy-to-consume data standards like Atom and JSON allow integration to be performed in a wide variety of ways.

### Introduction

Enabling WebSphere infrastructures to deliver feeds of data out into the fabric of the Internet will enable a wider range of clients access this data. The clients will no longer just be in the form of a Web browser. They may be in the form of a third-party computer system that may want to process a feed or a visually intuitive gadget or widget that has been embedded into desktop on operating systems like Windows Vista® or Mac OS.

By using the Web 2.0 FEP it is now possible to retain the existing business logic and data-persistent layers within WebSphere and then use capabilities of the Web 2.0 FEP to help surface the data using JSON, XML, or Atom formats.

## 13.3 Mobile devices

More and more people are using mobile devices to perform day-to-day tasks, such as diary management, mobile phone communication, playing music, and storing business data. The mobile market is booming, and with the introduction of the iPhone and new Google Android-based devices this will enable a new breed of applications to be developed. This types of devices will start to create a different way in which users will interact with a service, business, or enterprise. They will have different requirements around data access and feed management and may need to store data locally to allow the user access to information when not connected to the Internet.

The most common way to access a service from a mobile device is to use the embedded Web browser that is built into most modern mobile device platforms.

These browsers support the full range of Internet standards around HTML, DHTML, CSS, and JavaScript. This will enable them to offer a user a compelling Web 2.0 User Interface similar to a desktop browser.

The other area of user interface design on these platforms is to develop applications that take advantage of the native operating system. The iPhone has recently released an iPhone SDK, while Google is working on releasing the Google Android SDK. Windows Mobile® 6 offers some useful tools, and this is the platform that IBM supports with the Lotus Expeditor for Devices runtime that enables Eclipse eRCP-based and eSWT-based development. By using the features inside these native run times it is possible to interact with WebSphere middleware services that are exposed using the Web 2.0 Feature Pack.

### 13.3.1 Lotus Expeditor for devices

IBM Lotus Expeditor software is the IBM universal desktop or device client integration framework. It assists developers in integrating a wide variety of client and server applications into business mashups, or composite applications, to optimize the information that employees need to accelerate business processes. Lotus Expeditor software supports mobile computers, desktops, kiosks, and mobile device clients. It provides the flexibility of a service-oriented architecture (SOA) and a standards-based programming model from the OSGi alliance and the Eclipse foundation.

Expeditor supports the Java programming language and is based upon the Eclipse Rich Client Platform. It contains a rich set of services that enable the client to be integrated into wide variety of middleware platforms. This can include:

- ▶ Database services through JDBC
- ▶ Messaging services with JMS and Microbroker
- ▶ Web services through WSDL and JAX-RPC
- ▶ RESTful services through its local Web container

By using the local Web container you can build Web-based desktop and device solutions that use the power of the Web technologies but can also run locally on the device. This allows for a mediation layer to be added to a desktop or mobile device solution that can manage if the Internet connection is lost or is unavailable. This architecture is being used to help support a variety of different industry solutions from retail banking to retail distribution and chemical and petroleum solutions that are monitoring oil flow and pipelines. The use of Ajax technologies in the UI allows for common components to be written that can be delivered to the channel and platform that needs them. By using RESTful services the integration is seamless.

### 13.3.2 iPhone SDK

iPhone OS comprises the operating system and technologies that you use to run applications natively on iPhone and iPod touch devices. Although it shares a common heritage and many underlying technologies with Mac OS X, iPhone OS was designed to meet the needs of a mobile environment, whereas users' needs are slightly different. Existing Mac OS X developers will find many familiar technologies, but they will also find technologies that are available only on iPhone OS, such as the multi-touch interface and accelerometer support.

The iPhone SDK contains the code, information, and tools that you need to develop, test, run, debug, and tune applications for the iPhone OS. The Xcode tools have been updated to support development for the iPhone OS. In addition to providing the basic editing, compilation, and debugging environment for your code, Xcode also provides the launching point for testing your applications on an iPhone or iPod touch device. Xcode also lets you run applications in iPhone simulator, a platform that mimics the basic iPhone OS environment on your local Macintosh computer.

The iPhone SDK supports the creation of graphically oriented applications that run natively in iPhone OS. The applications that you create reside on the user's home page, along with the other system applications, such as photos, weather, and clock. After it is launched, aside from the kernel and a few low-level daemons, your application is the only application running on the system. While running, your application occupies the entire window and is the focus of the user's attention. And when the user presses the home button, your application quits and the system displays the home page again. Having the system to yourself is advantageous because it gives you full access to the underlying system resources. You can take advantage of built-in hardware such as the accelerometers, camera, and graphics hardware to run just your code.

Because the way users interact with iPhone and iPod touch devices is fundamentally different from the way that users interact with Mac OS X, the way in which you design your applications must also be different. In an iPhone application, there is no concept of separate document windows for displaying content. Instead, all of the application's data is displayed in a single window. This has led to the creation of new views and controls that allow you to present your application's data in an organized manner. In addition, many of the standard views and controls may behave a little differently from their Mac OS X counterparts. Most of these changes should be transparent, but some may require you to rethink the way in which you organize and present your data.

The event-handling model in iPhone OS also represents a significant departure from traditional desktop applications. Instead of relying on the traditional mouse and keyboard events, iPhone OS introduces the idea of touch events. A touch

event can occur at any time and in combination with one or more additional touch events. Touches can be used to detect simple interactions with content, such as selecting or dragging items, or they can be used to detect complex gestures and interactions, such as swipes or the pinch-open and pinch-close gestures (used, for example, to zoom in and out in the photos application).

Beyond considering the basic structure of your application, you need to think about how users will actually use it. iPhone applications should be clean and focused on what the user needs in the moment. Remember that users who are on-the-go want to get at information quickly and not spend a lot of time looking through several layers of windows. Providing a simple layout that highlights the key information that the user needs is important. For games and other fun applications, you should also consider how the users might want to interact with your application and take advantage of technologies such as the accelerometers and camera, where appropriate.

Given the rich capabilities of the iPhone, this platform is going to gain momentum within the enterprise users' space. This will involve solutions that take advantage of the iPhone and touch user interface and will interact with services that are hosted on the WebSphere platform. Using the Objective-C programming language and UI Kit widgets developers will be able to interact with middleware-based services and display data in a custom user interface on the iPhone device. Using the SQLite database local data will be able to be stored and the device used for offline and online solutions.

### 13.3.3 Android SDK

Android is a software stack for mobile devices that includes an operating system, middleware, and key applications. It has been designed by Google as an more open and alternative platform than the Apple-based iPhone solution. The Android runtime contains the following features:

- ▶ Application framework enabling reuse and replacement of components
- ▶ Dalvik virtual machine optimized for mobile devices
- ▶ Integrated browser based on the open source WebKit engine
- ▶ Optimized graphics powered by a custom 2D graphics library; 3D graphics based on the OpenGL ES 1.0 specification (hardware acceleration optional)
- ▶ SQLite for structured data storage
- ▶ Media support for common audio, video, and still image formats (MPEG4, H.264, MP3, AAC, AMR, JPG, PNG, GIF)
- ▶ GSM Telephony (hardware dependent)
- ▶ Bluetooth, EDGE, 3 G, and WiFi (hardware dependent)

- ▶ Camera, GPS, compass, and accelerometer (hardware dependent)
- ▶ Rich development environment including a device emulator, tools for debugging, memory and performance profiling, and a plug-in for the Eclipse IDE

Developers will have full access to the same framework APIs used by the core applications. The application architecture is designed to simplify the reuse of components. Any application can publish its capabilities, and any other application may then make use of those capabilities (subject to security constraints enforced by the framework). This same mechanism allows components to be replaced by the user. By using the programming APIs you will be able to integrate Google Android applications with WebSphere solutions using the Web 2.0 Feature pack.

## 13.4 Adobe AIR

Adobe AIR is a cross-operating system run time that lets developers combine HTML, Ajax, Adobe Flash, and Flex technologies to deploy rich Internet applications (RIAs) on the desktop.

Adobe AIR allows developers to use familiar tools such as Adobe Dreamweaver® CS3, Flex Builder™ 3, Flash CS3 Professional, or any text editor to build their applications and easily deliver a single application installer that works across operating systems.

AIR is gaining momentum due to the fact that Adobe has a large desktop and browser install base with Flex and Flash. Companies like eBay are delivering desktop clients that seamlessly integrate the client application with data services on the Internet. Demonstrating how Adobe AIR can be integrated with WebSphere middleware will help an organization to understand that existing assets that have been developed onto of the J2EE specification and that are running on an existing WebSphere Application Server infrastructure can be easily exposed for consumption by this exciting RIA platform.

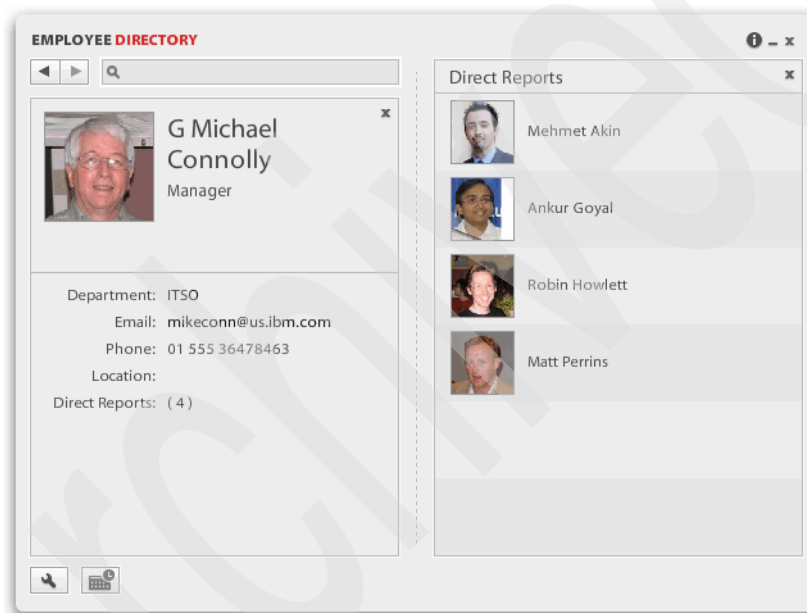
### 13.4.1 Scenario

In the following example we extend an existing Adobe AIR sample application and show how it can be integrated with a WebSphere Application Server application using the Web 2.0 Feature Pack. The scenario is an employee directory application that will retrieve a list of employees from a server and then cache them locally within an Adobe AIR database.

At initialization time the AIR application establishes a connection with its synchronization implementations. It then retrieves the data in a specific format and parses it into employee objects. These objects are then persisted into a simple database schema that is stored in the local file system. Subsequent invocations of the application then utilize the local database.

When the users perform searches for employees it will perform the queries against the local database and then display the results.

This section focuses on explaining how you can integrate a WebSphere middleware implementation consisting of a list of employee objects into the Adobe AIR client directory client.



*Figure 13-1 Adobe employee directory integrated with WebSphere Application Server*

The application is divided into two parts:

- ▶ The client-side Adobe AIR user interface that is built using Adobe Flex.
- ▶ The server-side data store for the employee data that can be accessed using a WebSphere Application Server and the Web 2.0 Feature Pack RPC Adapter.

## 13.4.2 Server

The server component for this application is going to be designed to demonstrate how to obtain data from the WebSphere Application Server and present it in a form that can be integrated with a client technology that is not part of the traditional J2EE request/response programming model.

The WebSphere application will offer a REST type entry point used by the Adobe AIR client to initialize the data for the employee directory application. To set up the server components from the application:

1. First create a dynamic Web project and define an Enterprise Application Project for the Web project to be registered with.
2. Create a project named `ITSOClientServices` and an EAR file named `ITSOClientServicesEAR`.
3. Configure the Web project to support the `RPCAdapter`. Do this by copying the following file from the `Web2fp` directory below the WebSphere installation to the `WEB-INF/lib` directory within the Web project.

*Table 13-1 Configuring the Web project to support the RPCAdapter*

Source: <was install>	Destination
Web2fep/optionalLibraries/RPCAdapter/commons-logging-1.0.4.jar	WEB-INF/lib
Web2fep/optionalLibraries/RPCAdapter/retroweaver-rt-2.0.jar	WEB-INF/lib
Web2fep/optionalLibraries/RPCAdapter/RPCAdapter.jar	WEB-INF/lib
Web2fep/optionalLibraries/JSON4J/JSON4J.jar	WEB-INF/lib

4. Once the Java Libraries required by the RPC Adapter are configured, you must configure the servlet for handling the REST-based requests.
5. Open the deployment descriptor for the Web project and add the servlet definitions shown in Example 13-1.

*Example 13-1 RPCAdapter servlet configuration for the Web project*

```
<servlet>
  <description>
  </description>
  <display-name>RPCAdapter</display-name>
  <servlet-name>RPCAdapter</servlet-name>
```

```

    <servlet-class>
        com.ibm.websphere.rpcadapter.RPCAdapter
    </servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>RPCAdapter</servlet-name>
    <url-pattern>/services</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>RPCAdapter</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>

```

---

6. Once you have completed editing the deployment descriptor you can save it.
7. The next step is to define the Plain Old Java Objects (POJOs) objects that will be used by the RPC Adapter to retrieve the list of employee objects.
8. Within the project src directory create a package called com.ibm.itso.services.model.
9. Within the com.ibm.itso.services.model directory create the POJO object named Employee.java that is shown in Example 13-2.

*Example 13-2 Employee.java POJO object*

---

```

package com.ibm.itso.services.model;

public class Employee {

    public String id;
    public String firstName="";
    public String lastName="";
    public String displayName="";
    public String title="";
    public String department="";
    public String managerId="";
    public String Employee="";
    public String email="";
    public String phone="";
    public String cellPhone="";
    public String phoneExtension="";
    public String deskLocation="";
    public String location="";
    public String city="";
    public String state="";
    public String postalCode="";
}

```



```

    public String countryCode="";

    public String getCalendarId() {
        return email;
    }

    public String toString() {
        return "Employee : " + displayName;
    }
}

```

---

10. Notice that the POJO does not yet have any getter or setters methods. Using the context menu on the Java editor select **Context Menu** → **Source** → **Generate Getters and Setters**.

11. You can now save the Employee.java POJO object.

The next step is to create an accessor bean that can be used to prepare the data returned from the REST style URL call:

1. In the com.ibm.itso.services directory create a POJO called EmployeeList.java.
2. Add the code shown in Example 13-3 to this Java class.

*Example 13-3 EmployeeList POJO that returns a set of employee objects*

---

```

package com.ibm.itso.services;

import java.util.ArrayList;

import com.ibm.itso.services.model.Employee;

public class EmployeeList {

    private ArrayList<Employee> employees;

    public EmployeeList() {

        employees = new ArrayList<Employee>();

        Employee emp = new Employee();
        emp.setFirstName("G Michael");
        emp.setLastName("Connolly");
        emp.setDisplayName("G Michael Connolly");
        emp.setEmail("mikeconn@us.ibm.com");
        emp.setId("mikeconn");
    }
}

```

```
emp.setPhone("01 555 36478463");
emp.setLocation("IBM Poughkeepsie");
emp.setTitle("Manager");
emp.setDepartment("ITS0");
employees.add(emp);

emp = new Employee();
emp.setFirstName("Matthew");
emp.setLastName("Perrins");
emp.setId("mperrins");
emp.setDisplayName("Matt Perrins");
emp.setEmail("matthew_perrins@uk.ibm.com");
emp.setPhone("44 2555 464778");
emp.setManagerId("mikeconn");
emp.setLocation("IBM Warwick");
emp.setTitle("Executive IT Specialist");
emp.setDepartment("ITS0");
employees.add(emp);

emp = new Employee();
emp.setFirstName("Mehment");
emp.setLastName("Akin");
emp.setId("makin");
emp.setDisplayName("Mehmet Akin");
emp.setEmail("mehmet.akin@ie.ibm.com");
emp.setPhone("01 555 981851931");
emp.setManagerId("mikeconn");
emp.setDepartment("ITS0");
emp.setLocation("IBM Dublin");
emp.setTitle("Software Engineer");
employees.add(emp);

emp = new Employee();
emp.setFirstName("Ankur");
emp.setLastName("Goyal");
emp.setId("agoyal");
emp.setDisplayName("Ankur Goyal");
emp.setEmail("ankurgoyal@in.ibm.com");
emp.setPhone("981851931");
emp.setPhone("44 555 464778");
emp.setManagerId("mikeconn");
emp.setDepartment("ITS0");
emp.setLocation("IBM Deli");
emp.setTitle("IT Specialist");
employees.add(emp);
```

```

        emp = new Employee();
        emp.setFirstName("Robin");
        emp.setLastName("Howlett");
        emp.setId("rhowlett");
        emp.setDisplayName("Robin Howlett");
        emp.setEmail("robin.howlett@ie.ibm.com");
        emp.setPhone("01 555 8393894");
        emp.setManagerId("mikeconn");
        emp.setDepartment("ITS0");
        emp.setLocation("IBM Dublin");
        emp.setTitle("Software Engineer");
        employees.add(emp);
    }
    public ArrayList list()
    {
        return employees;
    }
}

```

3. You can see that this bean creates an `ArrayList` of `employees`, and when each employee is created it is populated with data and added to the collection.

**Note:** You can use a variety of techniques to call back-end logic to dynamically generate the necessary objects needed. This example is only going to manually create a set of employee objects. This code could be replaced with a back-end service call.

The final stage of the server development is to configure the RPC Adapter.

4. Create a file called `RpcAdapterConfig.xml` in the `WEB-INF` directory of your dynamic Web project.
5. Edit the `RpcAdapterConfig.xml` file and include the statements shown in Example 13-4.

#### *Example 13-4 RPC configuration*

```

<?xml version="1.0" encoding="UTF-8"?>
<rpcAdapter

xmlns="http://www.ibm.com/xmlns/prod/websphere/featurepack/v6.1/RpcAdapterConfig"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <default-format>json</default-format>
    <services>
        <pojo>

```

```
<name>employees</name>

<implementation>com.ibm.itso.services.EmployeeList</implementation>
  <methods filter="whitelisting">
    <method>
      <name>list</name>
      <description>Retrieve a list of Employees.</description>
    </method>
  </methods>
</pojo>
</services>
</rpcAdapter>
```

---

You can see from this configuration that the `EmployeeList` bean has had its `list()` method added as a possible endpoint for the adapter.

6. Save the configuration and start the `ITSOClientServicesEAR` project within a WebSphere Application Service test environment.

You have now completed the setup of the server elements for the example. The final step is to test the RPC Adapter to check whether it is going to produce the necessary JSON to be used by the employee directory client.

1. Open up the browser with the following URL

`http://localhost:9080/ITSOClientServices/services/httprpc`

2. You should see the list of defined RPCAdapter service endpoints shown in Figure 13-2. Select the link to see the defined operations for the give RPC Adapter service.



Figure 13-2 RPC Adapter available services

3. You can now test the following URL in a browser:  
[http://localhost:9080/ITSOCClientServices/services/httprpc/employees/  
list](http://localhost:9080/ITSOCClientServices/services/httprpc/employees/list)

The mime type for the URL request is set to application/json. This causes a browser prompt to save the contents of the request as a file, as shown in Figure 13-3.

**Note:** To make it easier to see the contents of this request, when the Opening list dialog appears select **WordPad** as the default program to handle opening the contents.

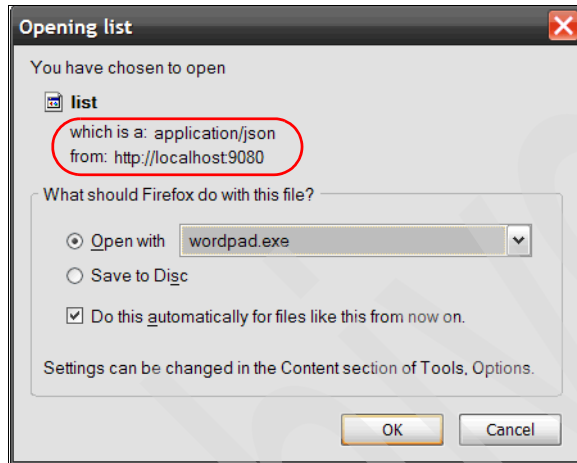


Figure 13-3 Configure Firefox to use WordPad to open application/json mime types

4. Click **OK** to save the settings. The file should then open in WordPad and show you the contents of the request, as shown in Figure 13-4.

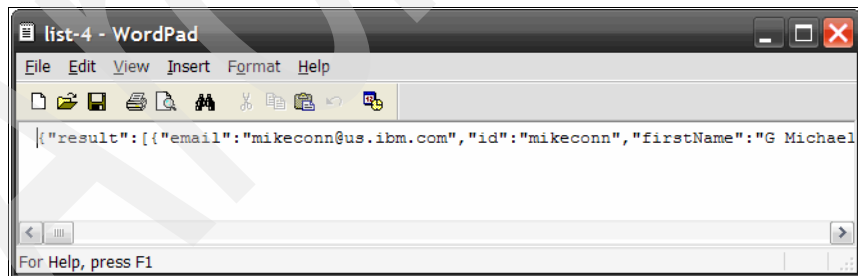


Figure 13-4 JSON contents generated from the RPC Adapter

This now concludes the setting up of the server elements for the employee directory sample.

### 13.4.3 Client

The employee directory client is a sample application provided by Adobe that installs into the Adobe AIR runtime on the desktop to showcase the power of a Rich Internet application. It also demonstrates some interesting RIA patterns, such as:

- ▶ The use of rich graphics for the display of employees and their relationships with their manager and teams.
- ▶ Data collection is acquired from a server when the client application is started. This data is then cached in a local database for use during subsequent invocations of the applications.

Using a comprehensive sample application that could be easily extended to make REST style calls to WebSphere could help demonstrate how the retrieval of employee data could be done by leveraging the Web 2.0 Feature Pack within WebSphere.

You can use some of these techniques to look at integrating WebSphere middleware applications into Adobe-based RIAs.

Follow the steps below to create a new version of the employee directory sample that integrates with WebSphere:

#### **Download the Adobe AIR runtime**

To download the Adobe AIR runtime:

1. The Employee Directory sample requires that the Adobe AIR runtime be installed on your desktop operating system.
2. Select the link below to download the installation package. Save a file called AdobeAIRInstaller.exe to your desktop:  
<http://get.adobe.com/air/>
3. Once the download has completed run the installation file.

## Install Adobe AIR runtime

To install:

1. The Adobe AIR installation will first prompt you with the license agreement screen shown in Figure 13-5.



Figure 13-5 Adobe AIR accept license panel

2. Select **I Agree** to start the Adobe AIR installation.
3. The installation process will display a progress panel, as shown in Figure 13-6.

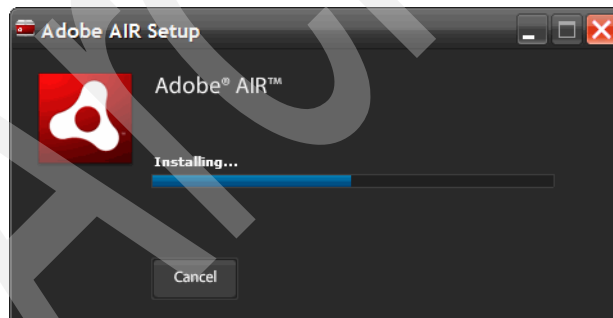


Figure 13-6 Adobe AIR installation progress



4. When the installation is complete you are presented with the display shown in Figure 13-7. Click **Finish**.

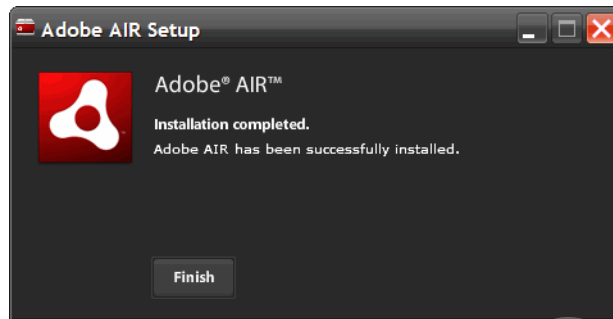


Figure 13-7 AIR setup complete

This completes the installation of the Adobe AIR runtime on your local operating system.

### Download Flex Builder 3

With the Adobe AIR runtime installed, the next step is to set up a development environment for the sample application.

**Note:** Adobe operates a model where the run time is free for installation, but the development tooling costs money. This sample was built using the 60-day try-and-buy version of Flex Builder 3.

1. Download the Flex Builder 3 Eclipse-based development tooling from the following URL:  
<http://www.adobe.com/cfusion/entitlement/index.cfm?e=flex3email>
2. Select the platform type to which you are installing the tooling, then click the **Download Now** button.
3. You will be asked to save the installation file. If you select Windows you will be presented with a default file name similar to FB3\_win.exe. Save the file to your desktop.

You now have the Flex Builder 3 tooling installation files downloaded.

## Install Flex Builder 3

To install:

1. Run the program FB3\_win.exe. This starts the installation process (on the Windows platform).
2. Follow the installation steps to complete the Flex Builder 3 Installation.

## Download employee directory sample

With both the AIR runtime and the Flex Builder tooling installed, the next step is to download the employee directory sample. You can use this sample as the base sample application to integrate with the WebSphere Application Server by using the WebSphere Web 2.0 Feature Pack.

Use the following URL to download the sample code to your desktop:

[http://download.macromedia.com/pub/developer/air/sample\\_apps/EmployeeDirectory.zip](http://download.macromedia.com/pub/developer/air/sample_apps/EmployeeDirectory.zip)

## Import employee directory into Flex Builder

Once the sample has been downloaded you need to import the package into the Flex Builder 3 development tooling:

1. Open the Flex Builder 3 development tooling as shown in Figure 13-8.

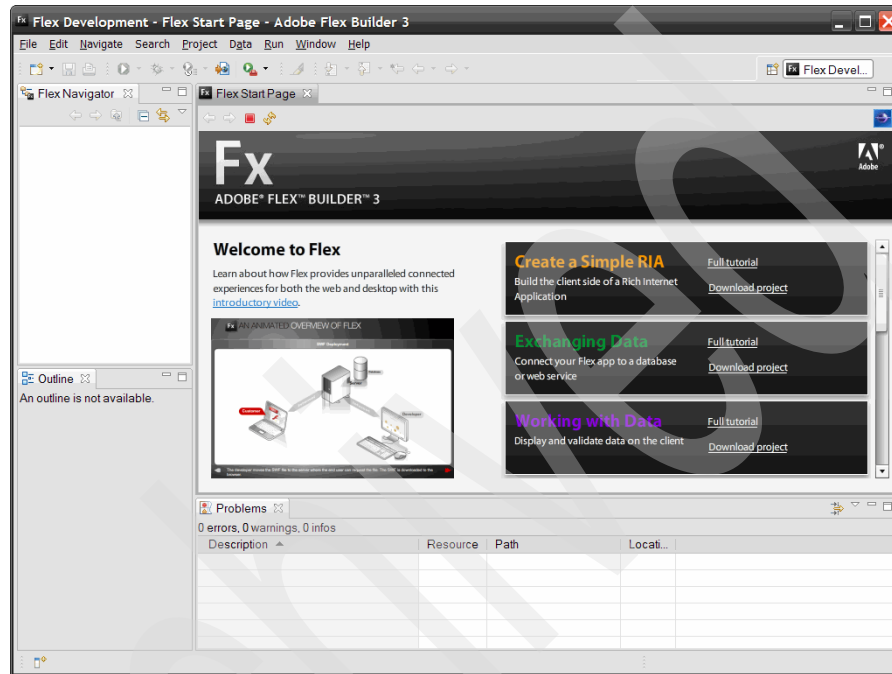


Figure 13-8 Adobe Flex Builder 3 Eclipse based development environment

2. Select **File** → **Import** → **Flex Project**.

3. Select **Archive file** and then the **Browse** button to select the **EmployeeDirectory.zip** file that you previously downloaded. See Figure 13-9.

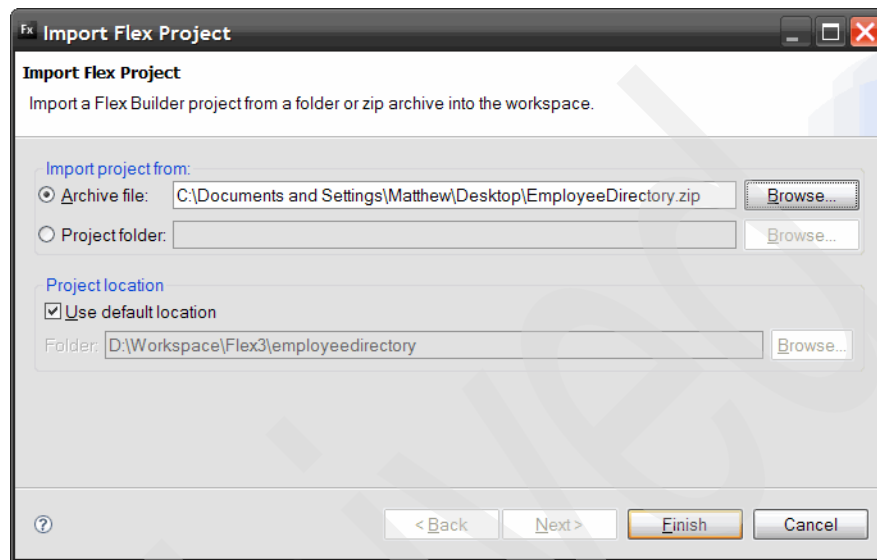


Figure 13-9 Import Flex Project

4. Click **Finish** to complete the import of the Flex project.

## Download JSON library

The corelib project is an ActionScript 3 Library that contains a number of classes and utilities for working with ActionScript 3. These include classes for JSON serialization, as well as general string, number, and date APIs.

1. Download the ActionScript library from the following URL and save it on your desktop:  
<http://as3corelib.googlecode.com/files/corelib-.90.zip>
2. Unzip the file corelib-.90.zip into a directory on your desktop.
3. Within the Flex Builder 3 tooling expand the **Employee Directory project** and expand the **src/com/adobe** directory.
4. Click **File** → **Import** → **Other**.

5. Expand the **General** folder, select **File System**, and click **Next**, as shown in Figure 13-10.

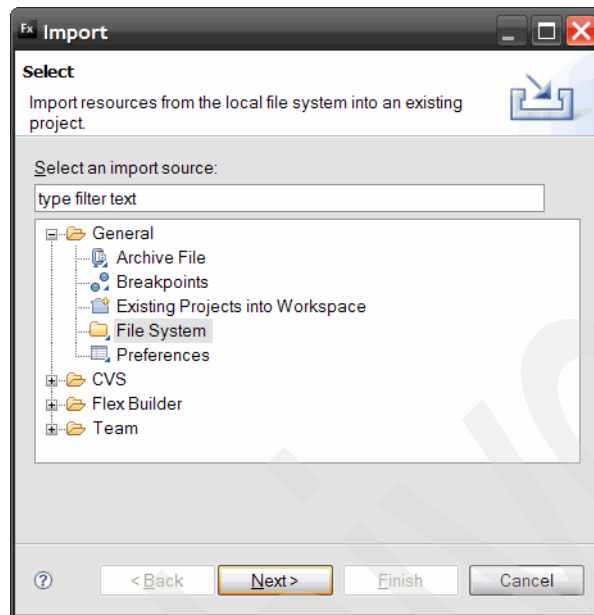


Figure 13-10 File import other

6. Click the **Browse** button and select the directory where you unzipped the corelib-.90.zip files.
7. Expand the directory structure until you are inside the corelib directory and then expand src/com/adobe directories.

8. Select the adobe directory and click **OK**, as shown in Figure 13-11.

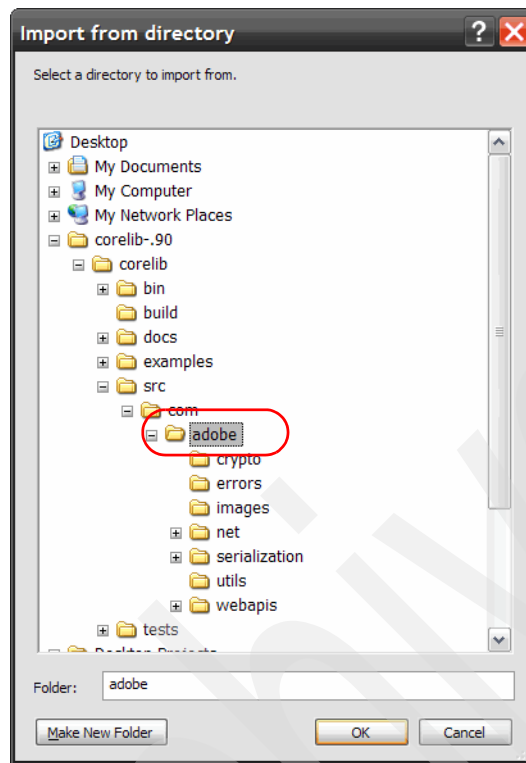


Figure 13-11 Import the serialization directory

9. Expand the **adobe** directory and then select **Serialization**.

10. Click the Into folder field's browse button and expand the **employeeedirectory/src/com/adobe** directory. You should have a dialog that looks similar to the one shown in Figure 13-12.

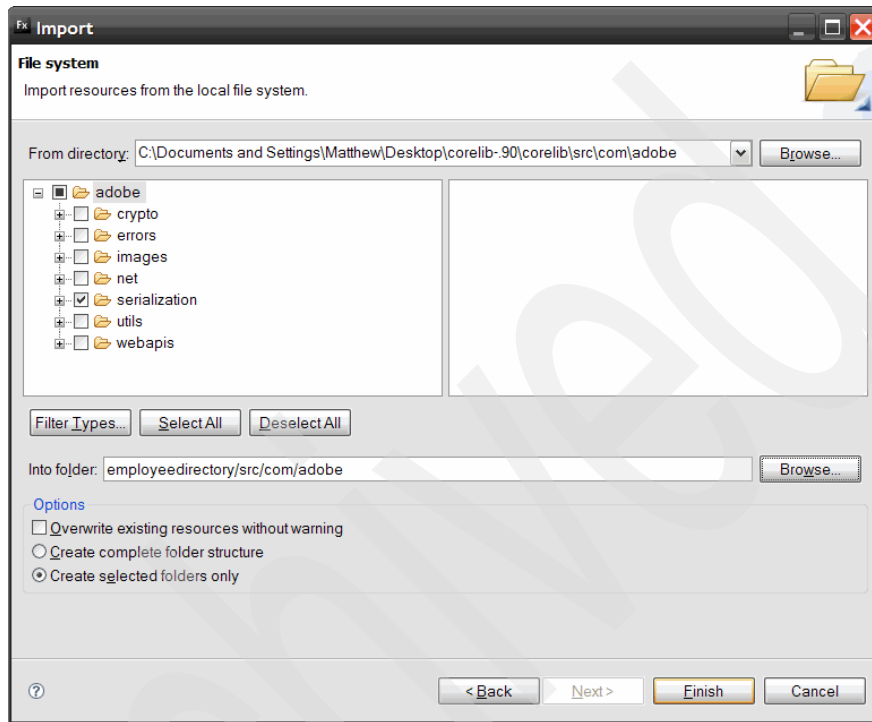


Figure 13-12 Import core library code into employee directory project

11. Click the **Finish** button to import just the serialization JSON ActionScript code.

You have now included the extra code need to support the serialization and deserialization of JSON format strings so that they can be used within Adobe AIR applications.

## Modify project

The following steps involve modifying the project code to support the delivery of the employee data located on the WebSphere Application Server. The employee directory application has been written in an extensible way and allows for the data to be delivered in different ways. The application contains a configuration file that allows the source handler to be specified:

1. Open up the config/employee\_directory\_config.xml file.
2. Add the configuration elements shown in Example 13-5 to the top of the configuration file.

*Example 13-5 Configuration for employee directory sample*

---

```
<!-- The URL to the employee data source. -->
<property key="employeeDataURL"
value="http://localhost:9080/ITSOClientServices/services/httprpc/employees/list" />
```

```
<!-- The parser class used for Employee objects. -->
<property key="employeeParserClass"
value="com.adobe.empdir.data.parsers.EmployeeJSONParser"/>
```

---

3. You need to comment out the existing configuration for the employeeDataURL and employeeParserClass. Use the XML `<!-- -->` notation for commenting out these elements.



## Add new EmployeeJSONParser

You have now specified the data URL that will be used during initialization of the employee directory application. The following steps involve the implementation of the EmployeeJSONParser:

1. Expand the `src/com/adobe/empdir/data/parsers` directory (Figure 13-13).
2. Click **File** → **New** → **ActionScript file**.

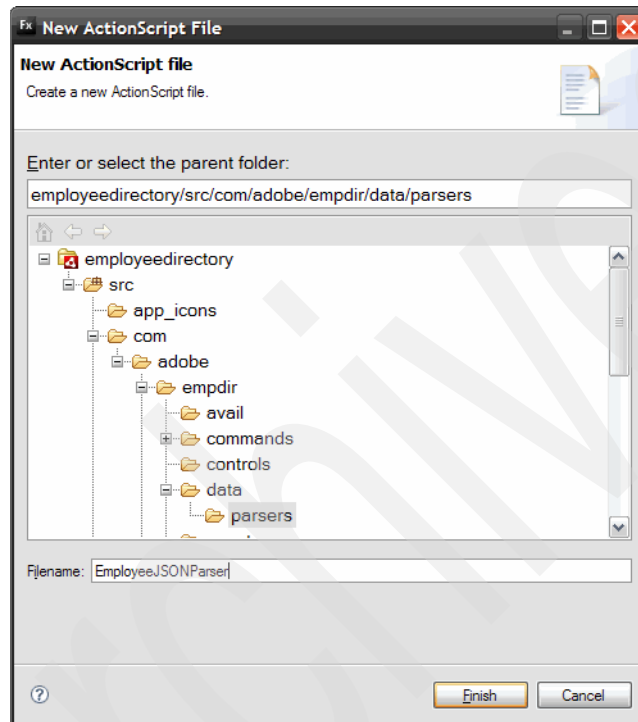


Figure 13-13 New ActionScript file

3. Make sure that the parent folder is set to `employeedirectory/src/com/adobe/empdir/data/parsers` and enter the name `EmployeeJSONParser` as the file name. Click **Finish** to create the file.

4. When the blank ActionScript file opens insert the code shown in Example 13-6 to complete the creation of the employee JSON parser functionality.

*Example 13-6 EmployeeJSONParser.as file*

---

```
package com.adobe.empdir.data.parsers
{
    import com.adobe.empdir.model.Employee;
    import com.adobe.serialization.json.JSON;

    public class EmployeeJSONParser implements IDataParser
    {
        public function parse( data:Object ) : Array
        {
            // Convert the Object to a String
            var sdata:String = data.toString();
            var obj:Object = JSON.decode(sdata) as Object;

            var employees:Array = obj.result;

            var results : Array = new Array();
            var employee : Employee;

            for each ( var entry:Object in employees )
            {
                employee = new Employee();

                employee.firstName = (entry.firstName != '' ?
entry.firstName : null);
                employee.lastName = (entry.lastName != '' ? entry.lastName
: null);
                employee.displayName = (entry.displayName != '' ?
entry.displayName : null);
                employee.id = (entry.id != '' ? entry.id : null);
                employee.title = (entry.title != '' ? entry.title : null);
                employee.email = (entry.email != '' ? entry.email : null);
                employee.managerId = (entry.managerId != '' ?
entry.managerId : null);
                employee.department = (entry.department != '' ?
entry.department : null);
                employee.location = (entry.location != '' ? entry.location
: null);
                employee.phone = (entry.phone != '' ? entry.phone : null);
            }
        }
    }
}
```

```

        employee.phoneExtension = (entry.phoneExtension != '' ?
entry.phoneExtension : null);
        employee.deskLocation = (entry.deskLocation != '' ?
entry.deskLocation : null);
        employee.city = (entry.city != '' ? entry.city : null);
        employee.state = (entry.state != '' ? entry.state : null);
        employee.countryCode = (entry.countryCode != '' ?
entry.countryCode : null);
        employee.postalCode = (entry.postalCode != '' ?
entry.postalCode : null);
        results.push( employee );
    }

    return results;
}
}
}

```

##### 5. Save the EmployeeJSONParser.as file.

The class structures implement the IDataParser interface and a method called parse( data:Object ). The method first retrieves the JSON string from the Data URL. It then invokes an instance of the JSON.decode method, which converts the JSON string into an ActionScript object. The result member is then used, as this contains a list of employee objects. A *for* loop is used to walk through all the employee object instances. These are then transposed into an instance of an employee object that is stored in a results array.

### Build and run

You are now ready to build and run. To complete the setup and testing of the sample application:

1. Ensure that the WebSphere Application Server that contains the ITSOClientServices project and the REST service for retrieving employee data is started.
2. To build and run the application select **Run** → **Run employeeDirectory**.
3. You are presented with the employee directory panel shown in Figure 13-14.



Figure 13-14 Employee directory front panel

4. Enter the letter M in the search field. The application will auto complete a drop-down list, as shown in Figure 13-15.

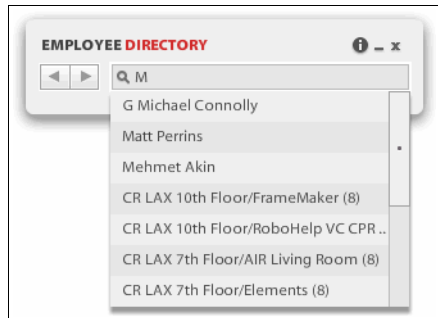


Figure 13-15 Employee directory search for letter M

5. Select employee **Matt Perrins** from the list.

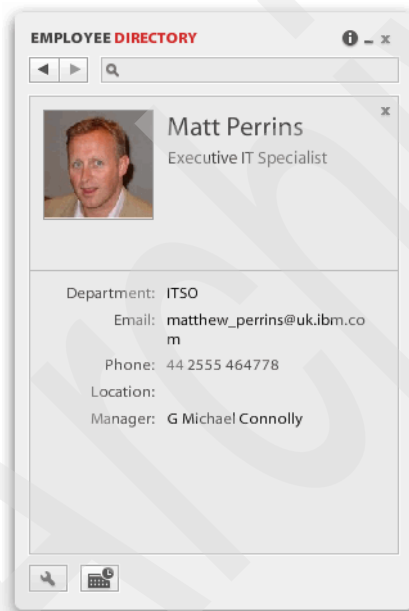


Figure 13-16 Employee display

6. Select the Manager name **G Michael Connolly**. You will see the employee directory change to the manager view.

7. Click **Direct Reports: (4)**. This displays the management structure, as shown in Figure 13-17.

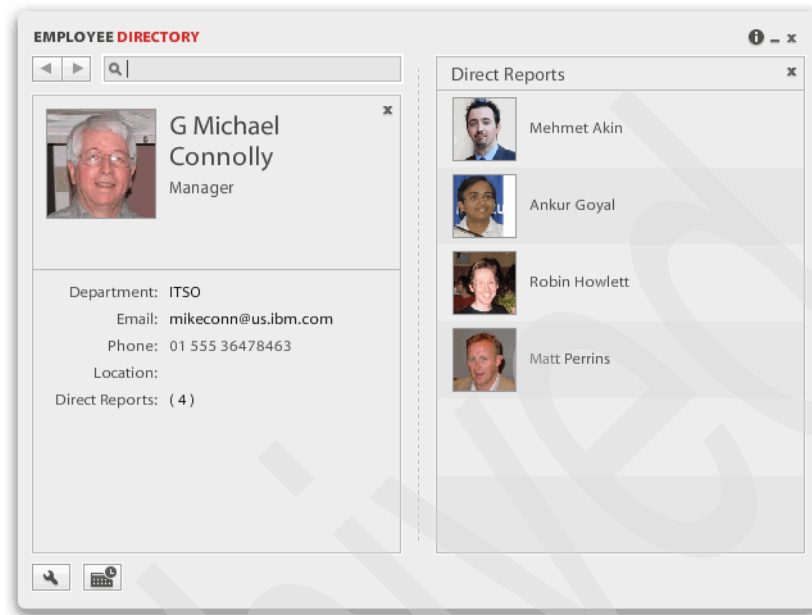


Figure 13-17 Employee managers panel showing direct reports

8. The final step is to list all the employees inside the organization. Select the **Department: ITSO**. The complete ITSO organization is displayed as shown in Figure 13-18

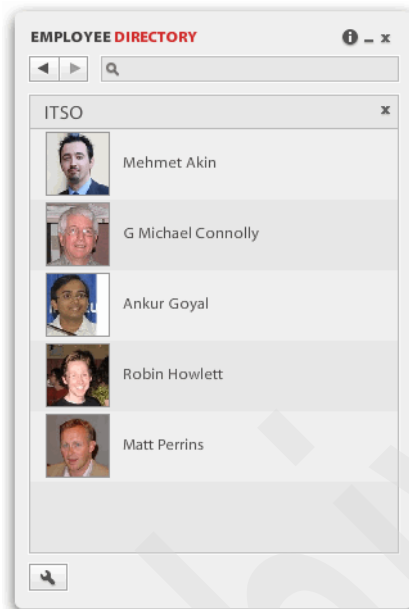


Figure 13-18 Employee directory organization panel

9. This completes the setup of the employee directory sample and integration into the WebSphere Web 2.0 Feature Pack.

**Note:** During initial application startup an `employeedirectory` file in the Documents and Settings/<User>/Application Data directory is created. To return the application to an initial state delete this directory. This forces the client to reconnect to the server and resynchronize the data.

## 13.5 Summary

This section provided an insight into how the Adobe AIR development runtime works and how it can be extended to integrate data hosted on the WebSphere Application Server.

You should now have an idea of the many rich client application frameworks available and how they use the various Web 2.0 technologies to integrate middleware business logic and data into the client application.

Archived





# **XML schema definitions**

Archived

## XML schema for RPCAdapterConfig.xml

Refer to the following XML schema to configure RPCAdapterConfig.xml appropriately for your solution application:

```
<?xml version="1.0" encoding="utf-8" ?>
<xsd:schema
  xmlns="http://www.ibm.com/xmlns/prod/websphere/featurepack/v6.1/RpcAdapterConfig"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.ibm.com/xmlns/prod/websphere/featurepack/v6.1/RpcAdapterConfig"
  elementFormDefault="qualified">
  <xsd:element name="services">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element maxOccurs="unbounded" ref="POJO" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="description" type="xsd:string" />
  <xsd:element name="scope">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Application" />
        <xsd:enumeration value="Session" />
        <xsd:enumeration value="Request" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
  <xsd:element name="validation-regex" type="xsd:string" />
  <xsd:element name="validator-ref" type="xsd:string" />
  <xsd:element name="default-format" type="xsd:string" />
  <xsd:element name="filtered" type="xsd:string" />
  <xsd:element name="recursive-object-support" type="xsd:string" />
  <xsd:element name="validator">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element minOccurs="1" ref="validation-regex" />
        <xsd:element minOccurs="1" ref="validation-class" />
      </xsd:sequence>
      <xsd:attribute name="id" type="xsd:string" use="required" />
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="bean-class" type="xsd:string" />
  <xsd:element name="converter-class" type="xsd:string" />
  <xsd:element name="converter">
```

```

        <xsd:complexType>
            <xsd:sequence>
                <xsd:element minOccurs="1" ref="bean-class" />
                <xsd:element minOccurs="1" ref="converter-class" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="implementation" type="xsd:string" />
    <xsd:element name="methods">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="method" maxOccurs="unbounded" />
            </xsd:sequence>
            <xsd:attribute name="filter" type="xsd:string" />
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="name" type="xsd:string" />
    <xsd:element name="parameters">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="parameter" maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="rpcAdapter">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="default-format" minOccurs="0" />
                <xsd:element minOccurs="0" ref="filtered" maxOccurs="1"
/>
                <xsd:element minOccurs="0"
ref="recursive-object-support" maxOccurs="1" />
                <xsd:element minOccurs="0" ref="converters"
maxOccurs="1" />
                <xsd:element minOccurs="0" ref="validators"
maxOccurs="1" />
                <xsd:element minOccurs="1" ref="services" maxOccurs="1"
/>
                <xsd:element minOccurs="0" ref="serialized-params"
maxOccurs="1" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="parameter">
        <xsd:complexType>

```

```

        <xsd:sequence>
            <xsd:element minOccurs="1" ref="name" />
            <xsd:element minOccurs="0" ref="description" />
            <xsd:element minOccurs="0" ref="validator-ref" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="http-method" type="xsd:string" />
<xsd:element name="validation-class" type="xsd:string" />
<xsd:element name="POJO">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="name" />
            <xsd:element ref="implementation" />
            <xsd:element minOccurs="0" ref="description" />
            <xsd:element ref="scope" minOccurs="0" />
            <xsd:element minOccurs="0" ref="validator-ref" />
            <xsd:element minOccurs="0" ref="methods" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="serialized-param-type" type="xsd:string" />
<xsd:element name="suppress" type="xsd:string" />
<xsd:element name="alias" type="xsd:string" />
<xsd:element name="suppressed-field" type="xsd:string" />
<xsd:element name="suppressed-fields">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element minOccurs="0" ref="suppressed-field"
maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="serialized-param">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element minOccurs="0" ref="serialized-param-type"
/>
            <xsd:element minOccurs="0" ref="suppress" />
            <xsd:element minOccurs="0" ref="alias" />
            <xsd:element minOccurs="0" ref="suppressed-fields" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="serialized-params">

```

```

        <xsd:complexType>
            <xsd:sequence>
                <xsd:element minOccurs="0" ref="serialized-param"
maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="add-to-session" type="xsd:string" />
    <xsd:element name="method">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="name" />
                <xsd:element minOccurs="0" ref="description" />
                <xsd:element minOccurs="0" ref="http-method" />
                <xsd:element minOccurs="0" ref="parameters" />
                <xsd:element minOccurs="0" ref="add-to-session" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="validators">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element minOccurs="0" ref="validator"
maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="converters">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element minOccurs="0" ref="converter"
maxOccurs="unbounded" />
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</xsd:schema>

```

## XML schema of profile-config.xml

Refer to the following XML schema to configure profile-config.xml as appropriate for your solution application:

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema
  xmlns:proxy="http://www.ibm.com/xmlns/prod/websphere/featurepack/v6.1/p
  roxy-config" xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.ibm.com/xmlns/prod/websphere/featurepack/
  v6.1/proxy-config">
  <xs:element name="policy">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="proxy:actions" />
        <xs:element ref="proxy:headers" />
        <xs:element ref="proxy:mime-types" />
        <xs:element ref="proxy:cookies" />
      </xs:sequence>
      <xs:attribute name="url" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="*" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
      <xs:attribute name="acf" use="optional">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value="none" />
          </xs:restriction>
        </xs:simpleType>
      </xs:attribute>
    </xs:complexType>
  </xs:element>
  <xs:element name="mime-types">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="proxy:mime-type" maxOccurs="unbounded"
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="mime-type">
    <xs:complexType />
  </xs:element>
```

```

<xs:element name="method">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="GET" />
      <xs:enumeration value="POST" />
      <xs:enumeration value="PUT" />
      <xs:enumeration value="HEAD" />
      <xs:enumeration value="DELETE" />
    </xs:restriction>
  </xs:simpleType>
</xs:element>
<xs:element name="mapping">
  <xs:complexType>
    <xs:attribute name="url">
      <xs:simpleType>
        <xs:restriction base="xs:anyURI" />
      </xs:simpleType>
    </xs:attribute>
    <xs:attribute name="contextpath" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:string" />
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>
</xs:element>
<xs:element name="headers">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="proxy:header" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="header">
  <xs:complexType />
</xs:element>
<xs:element name="cookies">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="proxy:cookie" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
<xs:element name="cookie">
  <xs:complexType />
</xs:element>

```

```
<xs:element name="actions">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="proxy:method" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```



## Additional material

This book refers to additional material that can be downloaded from the Internet as described below.

### Locating the Web material

The Web material associated with this book is available in softcopy on the Internet from the IBM Redbooks publication Web server. Point your Web browser at:

<ftp://www.redbooks.ibm.com/redbooks/SG247635>

Alternatively, you can go to the IBM Redbooks publication Web site at:

[ibm.com/redbooks](http://ibm.com/redbooks)

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks publication form number, SG24-7635.

## Using the Web material

The additional Web material that accompanies this book includes the following files:

<i>File name</i>	<i>Description</i>
<b>SG247635.zip</b>	Zipped code samples, readme file

## How to use the Web material

Create a subdirectory (folder) on your workstation, and unzip the contents of the Web material zip file into this folder. Open the readme file. This file contains the description of the accompanying files and the instructions for their use.

# Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

## IBM Redbooks publications

For information about ordering these publications, see “How to get Redbooks publications” on page 658. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *WebSphere Application Server Community Edition 2.0 User Guide*, SG24-7585
- ▶ *WebSphere Application Server Community Edition 2.0 User Guide*, SG24-7585
- ▶ *IBM WebSphere Application Server V6.1 Security Handbook*, SG24-6316
- ▶ *IBM WebSphere DataPower SOA Appliances Part II: Authentication and Authorization*, REDP-4364
- ▶ *IBM WebSphere DataPower SOA Appliances Part III: XML Security Guide*, REDP-4365
- ▶ *Portal Application Development Using WebSphere Portlet Factory*, SG24-7525

## Other publications

These publications are also relevant as further information sources:

- ▶ *The Long Tail: Why the Future of Business is Selling Less of More (Hardcover)*, Chris Anderson published by Hyperion books

## Online resources

These Web sites are also relevant as further information sources:

- ▶ O'Reilly, Tim. "Web 2.0: Compact Definition?" O'Reilly Radar, 1 October 2005  
<http://radar.oreilly.com/2005/10/Web-20-compact-definition.html>
- ▶ The Book of Dojo  
<http://dojotoolkit.org/book/export/html/589>
- ▶ Web 2.0 enhancements of the PlantsByWebSphere sample application  
[http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/topic/com.ibm.iea.wasfpWeb20/wasfpWeb20/6.0/Overview/PBW\\_J2EE\\_Ajax\\_enhancements.pdf](http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/topic/com.ibm.iea.wasfpWeb20/wasfpWeb20/6.0/Overview/PBW_J2EE_Ajax_enhancements.pdf)
- ▶ IBM Education Assistant: IBM WebSphere Application Server Feature Pack for Web 2.0  
[http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.wasfpWeb20/plugin\\_coverpage.html](http://publib.boulder.ibm.com/infocenter/ieduasst/v1r1m0/index.jsp?topic=/com.ibm.iea.wasfpWeb20/plugin_coverpage.html)
- ▶ ClassEasle: Web 2.0 Meets e-Learning, Jonathan Connolly  
<http://www.classeasle.com>
- ▶ Internationalizing Web applications using Dojo  
<http://www.ibm.com/developerworks/Web/library/wa-dojoo/>
- ▶ A look at the WebSphere Application Server Feature Pack for Web 2.0  
[http://www.ibm.com/developerworks/websphere/techjournal/0802\\_haverlock/0802\\_haverlock.html](http://www.ibm.com/developerworks/websphere/techjournal/0802_haverlock/0802_haverlock.html)
- ▶ Using the Dojo Toolkit with WebSphere Portal  
[http://www.ibm.com/developerworks/websphere/library/techarticles/0711\\_bishop/0711\\_bishop.html](http://www.ibm.com/developerworks/websphere/library/techarticles/0711_bishop/0711_bishop.html)
- ▶ Create reusable and redistributable components with Dojo and Ajax  
<http://www.ibm.com/developerworks/Web/library/wa-aj-components/index.html>
- ▶ Implementing client-side interportlet communication with Dojo and WebSphere Portal  
[http://www.ibm.com/developerworks/websphere/library/techarticles/0703\\_wallaker/0703\\_wallaker.html](http://www.ibm.com/developerworks/websphere/library/techarticles/0703_wallaker/0703_wallaker.html)
- ▶ Improve initial download time of your Dojo applications  
[http://www.ibm.com/developerworks/websphere/techjournal/0802\\_col\\_barcelona/0802\\_col\\_barcelona.html](http://www.ibm.com/developerworks/websphere/techjournal/0802_col_barcelona/0802_col_barcelona.html)

- ▶ Portlet development using REST services and the Dojo Toolkit  
[http://www.ibm.com/developerworks/websphere/library/techarticles/0803\\_lawless/0803\\_lawless.html](http://www.ibm.com/developerworks/websphere/library/techarticles/0803_lawless/0803_lawless.html)
- ▶ Develop HTML widgets with Dojo  
<http://www.ibm.com/developerworks/edu/wa-dw-wa-dojowidgets.html>
- ▶ Develop a Web application using Ajax with Dojo and DB2  
<http://www.ibm.com/developerworks/edu/dm-dw-dm-0702petrazickis-i.html>
- ▶ Creating Web 2.0 portals using Ajax and REST with WebSphere Portal  
[http://www.ibm.com/developerworks/websphere/library/chats/0705\\_hesmer/0705\\_hesmer.html](http://www.ibm.com/developerworks/websphere/library/chats/0705_hesmer/0705_hesmer.html)
- ▶ Build enterprise SOA Ajax clients with the Dojo toolkit and JSON-RPC  
[http://www.ibm.com/developerworks/websphere/library/techarticles/0606\\_barcia/0606\\_barcia.html](http://www.ibm.com/developerworks/websphere/library/techarticles/0606_barcia/0606_barcia.html)
- ▶ Web 2.0, Ajax, and REST in IBM WebSphere Portal  
[http://www.ibm.com/developerworks/websphere/library/presentations/0705\\_hesmer/0705\\_hesmer.html](http://www.ibm.com/developerworks/websphere/library/presentations/0705_hesmer/0705_hesmer.html)
- ▶ Use DWR, Java, and the Dojo Toolkit to integrate Java and JavaScript  
<http://www.ibm.com/developerworks/Web/library/wa-aj-ajaxpro3/>
- ▶ Develop a Dojo-based blog reader  
<http://www.ibm.com/developerworks/Web/library/wa-aj-basics2/>
- ▶ Ajax Security Basics  
<http://www.securityfocus.com/infocus/1868>
- ▶ Ajax Security  
<http://www.cgisecurity.com/ajax/>
- ▶ Top 10 Ajax Security Holes and Driving Factors  
<http://www.net-security.org/article.php?id=956>
- ▶ Are Ajax applications vulnerable to hack attacks?  
<http://www.acunetix.com/websitesecurity/ajax.htm>
- ▶ Ajax security — A reality check  
[http://searchsoftwarequality.techtarget.com/tip/0,289483,sid92\\_gci1219207,00.html](http://searchsoftwarequality.techtarget.com/tip/0,289483,sid92_gci1219207,00.html)

## How to get Redbooks publications

You can search for, view, or download Redbooks publications, Redpapers publications, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks publications, at this Web site:

[ibm.com/redbooks](http://ibm.com/redbooks)

## Help from IBM

IBM Support and downloads

[ibm.com/support](http://ibm.com/support)

IBM Global Services

[ibm.com/services](http://ibm.com/services)



# Building Dynamic Ajax Applications Using WebSphere Feature Pack for Web 2.0









# Building Dynamic Ajax Applications Using WebSphere Feature Pack for Web 2.0

## Web 2.0 features and benefits explained

## Ajax client runtime functions explored

## Sample implementations presented

This IBM Redbooks publication demonstrates techniques and technologies available through the WAS Feature Pack for Web 2.0 for building dynamic, next-generation Web applications. It covers the three main sub-components including:

- ▶ Connecting to SOA services from a Ajax using lightweight protocols REST and JSON
- ▶ Extending Enterprise Messaging to the Web using Ajax Messaging
- ▶ Speeding up Ajax application time to market using the Ajax Development Toolkit featuring Dojo

Web 2.0 FEP on WAS CE 2.0, WAS 6.1, and WAS 6.0.2 are supported.

## INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

## BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

**For more information:**  
[ibm.com/redbooks](http://ibm.com/redbooks)