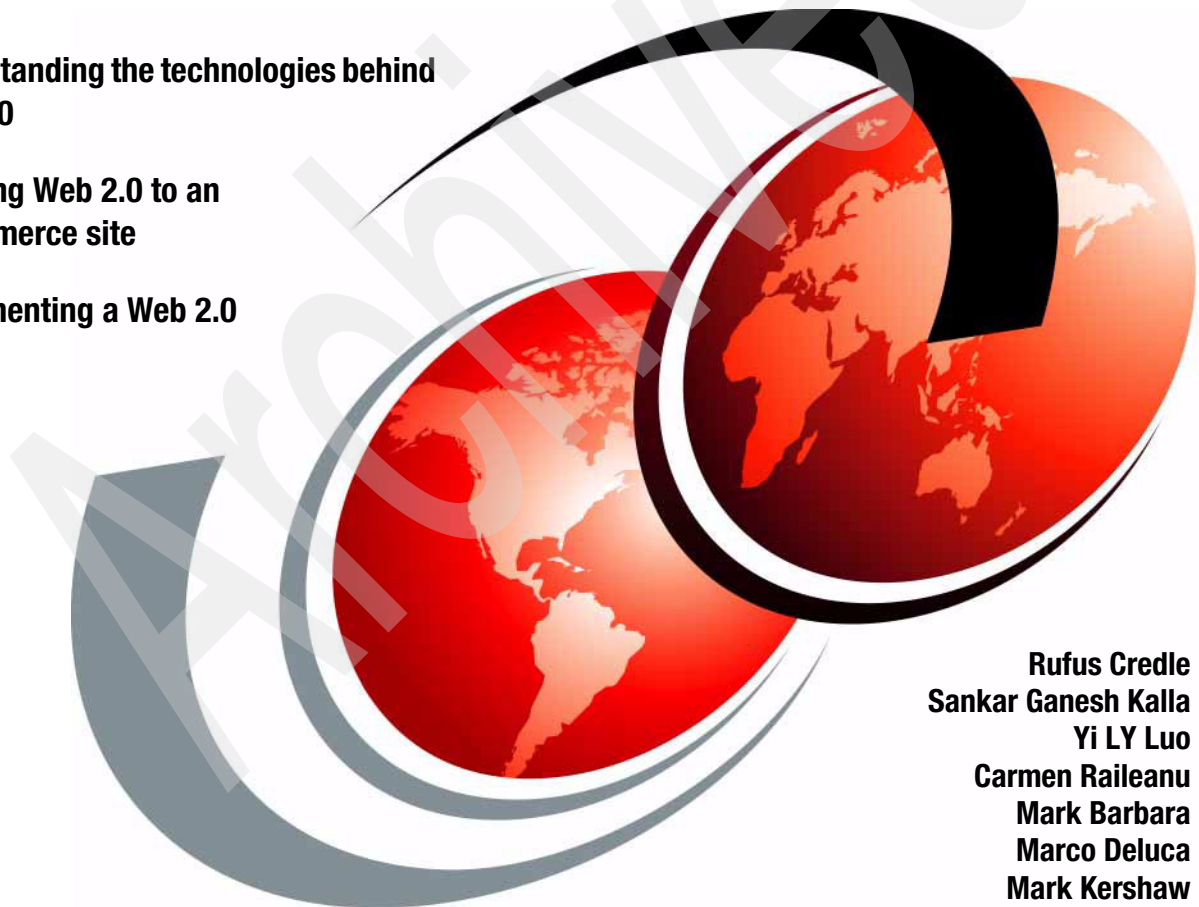


WebSphere Commerce Best Practices in Web 2.0 Store

Understanding the technologies behind
Web 2.0

Applying Web 2.0 to an
e-commerce site

Implementing a Web 2.0
Store



Rufus Credle
Sankar Ganesh Kalla
Yi LY Luo
Carmen Raileanu
Mark Barbara
Marco Deluca
Mark Kershaw



International Technical Support Organization

WebSphere Commerce Best Practices in Web 2.0 Store

April 2009

Archived

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

First Edition (April 2009)

This edition applies to WebSphere Commerce 6.1, Web 2.0, and Web 2.0 store.

© Copyright International Business Machines Corporation 2009. 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

Notices	vii
Trademarks	viii
Preface	ix
The team that wrote this book	ix
Become a published author	xi
Comments welcome	xii
Chapter 1. WebSphere Commerce Web 2.0 Store solution	1
1.1 Introduction to Web 2.0	2
1.1.1 The key concepts of Web 2.0	3
1.1.2 Technologies behind Web 2.0	4
1.1.3 Web 2.0 and service-oriented architecture	6
1.1.4 Rich Internet applications	8
1.2 Applying Web 2.0 to an e-commerce site	10
1.3 :WebSphere Commerce Web 2.0 Store solution	10
1.3.1 Introduction to the Madisons Store	10
1.3.2 Simplifying customization	12
1.4 Applying Dojo and Ajax to the Madisons Store	12
1.4.1 Architectural overview	13
1.4.2 Common interaction	14
1.5 Madisons Store Feature Pack 5 features	16
Chapter 2. WebSphere Commerce Web 2.0 Store features and benefits	17
2.1 Interactivity for user and benefits to clients	18
2.1.1 Rich shopping experience	18
2.1.2 Accessibility	24
2.1.3 Reducing the total cost of implementation	27
2.2 Model View Controller design pattern in Web 2.0	30
2.2.1 Model View Controller design pattern	30
2.2.2 How MVC applies for Web 2.0	32
2.3 Web 2.0 Store (FEP2) features and benefits	37
2.3.1 Single store supporting Web 1.0 and Web 2.0	38
2.3.2 Madisons Store Change Flow function options	38
2.3.3 Accessibility issues	40
2.3.4 Browser backward and forward buttons	40
2.3.5 Tooltips	40
2.3.6 The drag feature	41
2.3.7 Mini Shop cart	42

2.3.8 Quick Info panel	45
2.3.9 Scrollability widget.	46
2.3.10 Fast Finder	46
2.4 Web 2.0 application concepts	49
2.4.1 Overview of the Dojo toolkit	50
2.4.2 Including the Dojo library.	51
2.4.3 Dojo core library	52
2.4.4 XHR.	70
2.4.5 The Dojo Widget Library	79
Chapter 3. Design and implement your Web 2.0 Store	93
3.1 WebSphere Commerce Web 2.0 Store structure	94
3.1.1 WebSphere Commerce package structure for Web 2.0 Store	94
3.1.2 File content structure.	95
3.1.3 Page layout structure	102
3.2 WebSphere Commerce Ajax framework for Dojo	105
3.2.1 Getting familiar with the Ajax Framework extension	106
3.2.2 The four scenarios	118
3.3 WebSphere Commerce Framework for Dojo widgets	134
3.3.1 The Dojo widgets used for Web 2.0 Store	135
3.3.2 Implementing your own Dojo widgets	171
3.3.3 How to customize the Dojo widgets	174
3.4 Introduce Web 2.0 features into your Web 1.0 Store	183
3.4.1 Adding a Web 2.0 feature in your Web 1.0 Store	183
Chapter 4. Testing and debugging your Web 2.0 Store	195
4.1 Testing your Web 2.0 Store.	196
4.1.1 Functional testing	196
4.1.2 Usability testing	200
4.1.3 Performance testing	203
4.2 Debugging your Web 2.0 Store	210
4.2.1 Web 2.0 interactions	210
4.2.2 Breaking down the possible problem areas	211
4.2.3 Tools to debug and isolate problems	212
4.2.4 Identifying and isolating the problem area	215
4.2.5 The divide and conquer method	217
4.2.6 Debugging a sample problem	218
4.3 Optimizing Web 2.0 Store	223
4.3.1 Page weight review and performance tips	224
4.3.2 General performance tips	230
4.3.3 Advanced UI performance tips	232
4.3.4 Efficiently handling requests on client side	234
Appendix A. Additional material	237

Locating the Web material	237
Using the Web material	238
How to use the Web material	238
Related publications	239
IBM Redbooks	239
Online resources	239
How to get Redbooks	241
Help from IBM	242
Index	243

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 on 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:


BladeCenter®

DB2®

developerWorks®

IBM®

Redbooks®

Redbooks (logo) ®

System x®

WebSphere®

The following terms are trademarks of other companies:

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

Internet Explorer, Windows, and the Windows logo are trademarks of Microsoft Corporation 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 Web 2.0 Store (FEP2) and demonstrates how Web 2.0 is applied to an e-commerce site. In addition, you will gain an understanding of how Dojo and Ajax are applied to the Madisons Store (FEP5).

This book emphasizes some of the client benefits and describes the features of the new Web 2.0 Store solution of WebSphere® Commerce. This book discusses the Model View Controller (MVC) design pattern for Web 2.0 and provide guidance to developers who are building and customizing Ajax-based applications.

This book explains how to implement your own Web 2.0 Store using Ajax and Dojo and on the basis of the Madisons store. Developers can build their own Web 2.0 Store more efficiently (for example, quickly designing, developing, testing, analyzing, and deploying high-quality Web 2.0 Stores).

This book describes the best practices for testing and debugging your Web 2.0 Store functionality, and also presents the Web 2.0 Store optimization best practice. This book is targeted at WebSphere Commerce developers, WebSphere Commerce architects, and technical sales specialists.

Throughout this Redbooks publication, Web 2.0 Store pertains to the use of FEP2 and Madisons Store pertains to FEP5.

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, Raleigh Center.

Rufus Credle is a Certified Consulting IT Specialist at the ITSO, Raleigh Center. In his role as Project Leader, he conducts residencies and develops IBM Redbooks about network operating systems, ERP solutions, voice technology, high availability and clustering solutions, Web application servers, pervasive computing, IBM and OEM e-business applications, IBM System x®, IBM x-Series, and IBM BladeCenter®. Rufus' various positions during his IBM career have included assignments in administration and asset management, systems engineering, sales and marketing, and IT services. He holds a BS degree in business management from Saint Augustine's College. Rufus has been employed at IBM for 28 years.

Sankar Ganesh Kalla is a WebSphere Commerce Developer in India. He has four years of experience in e-Commerce field. His areas of expertise include Java™, J2EE™, AJAX, Dojo, and flex. He has written extensively on WebSphere Commerce store model related articles. He holds a degree in B.Tech Computer Science from JNTU Hyderabad, India.

Yi LY Luo is a technical lead of WebSphere Commerce development team in China Development Lab. He has 3 years of experience in e-Commerce field. He has worked at IBM for 3 years. His areas of expertise include e-Commerce solutions, J2EE development, and Web 2.0 technology and development. He got his master of science degree from Wuhan University, China.

Carmen Raileanu is an application developer working for Bestware Solution in Romania. She has seven years of experience in J2EE technologies programming field. She has professional certifications in Sun™ Certified Programmer, IBM Solution Developer and IBM Enterprise Application Developer and has completed the following IBM training: IBM DB2® Content Manager, WebSphere MQ, and WebSphere Application Server. Her areas of expertise include consulting and implementing of e-commerce solutions on IBM WebSphere Commerce, enterprise Web applications on J2EE platforms, Internet banking on J2EE platforms, and document management solutions on the IBM DB2 Content Manager platform. She holds a master degree in Computer Science at University of Constanta, Romania

Mark Barbara is the team lead of the WebSphere Commerce Level 2 support team at the IBM Toronto Lab. He has been working with WebSphere Commerce since starting at IBM in 2004. He has a wide depth of knowledge when it comes to debugging and troubleshooting problems for WebSphere Commerce, and the related IBM stack of products. His areas of expertise include Commerce Portal integrations as well as debugging site performance and stability problems for WebSphere Commerce. He graduated with distinction from the University of Toronto, earning his Bachelor of Science degree in Computer Science, specializing in Software Engineering, and majoring in Economics.

Marco Deluca is a Business Solution Architect team lead with the WebSphere Commerce Solution Enablement team at the IBM Toronto lab. He brings over ten years of progressive experience to the enterprise software domain with technical specialties including Web 2.0 solutions, Search Engine Optimization (SEO) and J2EE-related software development. He also has a considerable listing of e-Commerce published material in his area of expertise.

Mark Kershaw is a member of the Performance Delivery team of IBM Software Services for WebSphere. His areas of expertise center around load testing and performance tuning of large-scale WebSphere Commerce applications. He has experience with both of these activities on Web 2.0 implementations. Prior to this book, he co-authored works specific to the WebSphere Commerce Web 2.0 Store model. Before joining IBM, he obtained a BSc in Computer Science from the University of Calgary, Canada.

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

Tamikia Barrow, Margaret Ticknor
International Technical Support Organization, Raleigh Center

Jacob T. Vandergoot, WebSphere Commerce Architect, WebSphere Commerce Development
IBM Canada

Karson Ng, WebSphere Commerce Development (B2B Solutions)
IBM Canada

Woodward Aichner, Managing Consultant - SWG Team Supporting GBS
eCommerce Solutions Delivery
IBM Research Triangle Park, NC

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

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 in one of the following ways:

- ▶ Use the online **Contact us** review Redbooks form found at:

ibm.com/redbooks

- ▶ Send your comments in an e-mail to:

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

WebSphere Commerce Web 2.0 Store solution

This chapter introduces you to Web 2.0 and demonstrates how Web 2.0 is applied to an e-commerce site. In addition, you will gain an understanding how Dojo and Ajax are applied to the Web 2.0 Starter Store.

1.1 Introduction to Web 2.0

Web 2.0 represents a new wave in business innovation concerning how enterprises are trying to exploit the current generation of Internet technologies. The IBM developerWorks® article *Enterprise Web 2.0, Part 1: Web 2.0 -- Catching a wave of business innovation* defines Web 2.0 components in the following manner:

A set of patterns through which technology is currently being used to create and support business models using Internet technologies.

You can download this article from the following Web page:

<http://www.ibm.com/developerworks/webservices/library/ws-enterprise1/>

Web 2.0 represents an evolution of Web development through the creation of a rich UI. The current interest in Web 2.0 technology has arisen due to a culmination of economic and technology directions that make it a source of business innovation.

The following list explains a few of the paths that led to Web 2.0:

- ▶ Global participation in collaborative, richly featured communication technologies
Users need to communicate and exchange information with each other without leaving their own Web-based applications.
- ▶ A global trend of participation in collaborative communication technologies
Studies show that from the more than one billion people that are now online, the percentage of people interacting with each other socially and economically in expanding networks, rather than using the Web simply to shop or access information, is growing every day.
- ▶ Increasing capabilities and falling costs for the manufacturers of personalized or customized products
Anything from clothes to devices to automobiles can be customized or personalized with minimum addition to its normal price point, rather than requiring expensive costs and customizations.
- ▶ A continuous trend in business optimization
Exposure of business enterprise services through service-oriented architecture (SOA) are all the way out to the browser.
- ▶ Optimization of UI and interaction with products
Dynamic and easy-to-use UIs allow users to interact with Web page content.

- ▶ Simple and easy-to-use Web sites and rich user experiences on the Internet
Users need to find themselves interacting with features provided by the designer and getting exactly what they want. We need to remove unnecessary components that might confuse users.

This section introduces the following topics:

- ▶ The key concepts of Web 2.0
- ▶ Technologies behind Web 2.0
- ▶ Web 2.0 and SOA
- ▶ Rich Internet Applications (RIA)

1.1.1 The key concepts of Web 2.0

The following terms are key concepts in Web 2.0 technology.

Collaboration

The idea of the Internet today is to provide a communication medium in which users from all around the world can participate and interact. Users want Web sites with which they can interact.

Personal profiles, blogs, and wikis allow people to communicate without understanding underlying Web and browser technologies. Using these technologies, people can share their interests and expertise and create collaborative content.

Syndication feeds and widgets

Syndicated feeds, simple streams of stories, and information formatted in Extensible Markup Language (XML) according to the Really Simple Syndication (RSS) or Atom protocols enable sites to carry a vast array of content and let users create their own content. Feeds are services created according to Representational state transfer (REST) principles (a set of services scalable and easy to use).

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.

Mashups

Mashup applications let users easily mix function and content from many sources into new applications. Users use mashup applications to combine external information such as news, events, or weather to mix with their own data, putting all information in one place.

1.1.2 Technologies behind Web 2.0

This section discusses technologies found in Web 2.0. See Figure 1-1 for a graphic representation of these technologies.

- ▶ Ajax as the model for a rich client
- ▶ Ajax libraries (for example, Dojo)
- ▶ JSON and XML as the data interchange format
- ▶ REST and Atom, used to form the basis of the service invocation model
- ▶ Dojo toolkit that provides an API for developers of Ajax applications

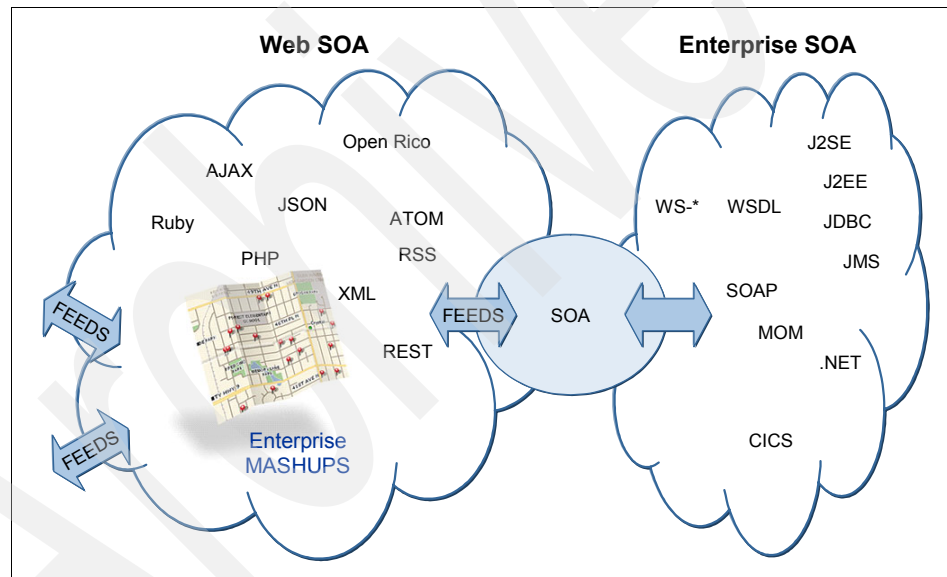


Figure 1-1 Web 2.0 technologies

Ajax

Asynchronous JavaScript™ and XML (Ajax) is a development technique for creating interactive Web applications. Data, content, and design are merged into a seamless whole. These techniques give the user of an Ajax-based Web application increased interactivity with the Web page content. This is achieved by exchanging small amounts of data with the server asynchronously using

JavaScript to communicate with a server. This means the entire Web page is not reloaded in a request or 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. The XMLHttpRequest object supports any text-based format, including XML. It is used for transferring data between client and server over both HTTP and HTTPS protocols.

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 Document Object Model (DOM).

Atom feeds and JSON

Atom feeds and JavaScript Object Notation (JSON) are technologies used to transport information from the services to the browser.

The Atom feed technology allows a user to create, update, and subscribe to resources. The Atom Syndication Format is an XML language used for Web feeds, while the Atom Publishing Protocol (also known as 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, summaries, or links to content on a Web site, along with various metadata.

JSON is a text-based human-readable format for representing simple data structures and associative arrays (called objects). 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.

REST and RESTful

One of the characteristics of a Web 2.0 application is that it represents a set of services that can be asynchronously invoked and merged with other services into a dynamic combination.

Ajax is one of the technologies that offers this kind of behavior for a Web application. An Ajax Web page separates presentation (list, trees, and so forth) from content (XML documents asynchronously fetched using JavaScript) and populates the presentation based on user actions on the page.

SOA containing the Representational State Transfer (REST) component, referred to RESTful SOA, is another technology that offers the possibility to extend some services out to the browser. RESTful SOA refers to delivering simple XML Web services delivered over HTTP using the REST approach. REST is a key component in an implementation of a RESTful SOA, and enables the creation of Web services that are simple to implement, build for reuse, and are scalable. We can build on the RESTful service model and add new metaphors like feeds.

The benefits of this technology are as follows:

- ▶ Allows building interactive Web pages
- ▶ Allows creating mashups from your published services and those of others
- ▶ Allows contacting customers in new ways, such as blogs and wikis
- ▶ Reduces the costs of implementation in terms of money and time

Dojo

Dojo is an open source JavaScript toolkit that provides an API that makes developing Ajax applications easy. It contains an API for client/server communication and a set of widgets ready to be used in Web pages. Section 2.4, “Web 2.0 application concepts” on page 49 provides more information on the components and widgets of Dojo Toolkit.

1.1.3 Web 2.0 and service-oriented architecture

SOA technologies in the form of Web services and WSDL are focused on supporting computer-to-computer integration. They support a wide variety of capabilities that make these the right solutions for enterprise integration within an Enterprise Service Bus architecture. One of the reasons that SOAP and WSDL are not more widely adopted is that Web browsers do not natively support these two specifications. Enterprises must focus on how they can release their assets on the Internet.

We can now view the Web as a set of services that can be asynchronously invoked, merged with other services, and rewoven into new, dynamic combinations. This approach is one of the defining characteristics of Web 2.0.

Ajax separates presentation and content into a basic page that contains the presentation (lists, trees, and so forth) and a set of XML documents that are asynchronously fetched (using JavaScript) and used to populate the presentation based upon user actions. Meanwhile, RESTful SOA is taking care of the services extended out to the browser. RESTful services and feeds based on protocols like RSS and Atom serve as good examples of well-defined SOA services.

Web 2.0

Web 2.0 applications are a combination of services, feeds, and widgets enabled through SOA. Web 2.0 has gained such a wide adoption because it allows developers to focus on solutions using easy-to-use concepts and technologies. Enabling feeds of data to be accessed in a RESTful way, enables business enterprise solutions to integrate with others.

A good example of this is Google Maps, used by millions of Web sites and developers. Google has managed to thread this service into the Internet and allow it to be consumed into a wide variety of Web 2.0 solutions.

The advantages of a Web 2.0 solution are as follows:

- ▶ It focuses on how a business enterprise can release their assets on the Internet.
- ▶ It uses established standards included in leading browsers already on consumers desktops, laptops, and mobile devices.
- ▶ It deploys data and logic into the Internet as services that can be easily invoked, upgraded, and deployed, which offers a high reuse.
- ▶ Software is being developed as a collaboration medium for the community. It uses a variety of social networking tools to enable quick feedback and comments.
- ▶ Simple programming models, languages, runtimes, and specifications are bases for Web 2.0 applications. REST and JSON are perfect examples of patterns and standards that provide enough of what most service developers need.

SOA provides easy access to the reusable business tasks in your enterprise. It allows combinations of different information from inside and outside business enterprises into services. REST and Web 2.0 provide a layer on top of this, making these services available to customers and users on the Internet. Mashup is one of the Web design patterns used to combine data from more than one source into a single integrated tool based on ready and easily accessible SOA services. In addition, you can allow customers to provide content in a way that is consistent with your business application by controlling the access to your services.

1.1.4 Rich Internet applications

Rich Internet applications (RIA) are Web applications that offer functionality and features similar to a desktop application. RIAs typically transfer the processing necessary for the UI to the Web client, but keep the bulk of the data on the application server. They can adapt to network connectivity and store data locally if needed. They also use the latest UI technologies to present a competing easy to use interface. Traditional Web applications are based on a client-server architecture using a thin client. In this design, most processing is performed on the server. 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 over time to accommodate these techniques, so it is difficult to draw a strict line between what constitutes an RIA and what does not. All RIAs, however, 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 UI, and for handling server communication.

Although developing applications to run in a Web browser is more difficult than developing traditional desktop applications, the extra effort is usually justified because of the following advantages:

- ▶ Local installation is not required. The application resides on the server.
- ▶ Updates and upgrades 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:

- ▶ Greater control

RIAs move some elements of the Model View Controller (MVC) design pattern from the server to the client runtime. 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 on when and how much data to retrieve and what to cache and when to make a decision to get more. This pattern is common in RIAs that use a wide variety of data volumes. For more information on MVC, see 2.2.1, "Model View Controller design pattern" on page 30.

► Richer functionality

RIAs offer UI behaviors not normally available using 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 the following examples:

- Drag and drop (DnD)
- Using a slider to change data
- Calculations performed only by the client, which 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 that the Web application server does not need to 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 clicking on 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 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 requirement 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 is used.

1.2 Applying Web 2.0 to an e-commerce site

Today, Web 2.0 e-commerce sites are using RIA technology to create a more usable and more efficient shopping experience for users, one that cannot be achieved with Web 1.0.

Retailers are incorporating social computing into e-marketing strategies to bring shoppers together by giving them a forum to exchange ideas and interact with the retailer on a personal level. For most retailers, their starting point for a social computing strategy is on merchandising. It begins with incorporating the product reviews in the e-commerce sites. Retailers like BassPro, Home Depot, and Discovery have expanded to the next level of product discussion by incorporating blogs and forums into their e-commerce sites to give both the business and shoppers a personality or voice to connect with others.

1.3 WebSphere Commerce Web 2.0 Store solution

WebSphere Commerce provides a starter store, known as *Madisons Store*, that illustrates the best of breed online shopping experience as described by Web 2.0. RIA is used to create a more usable and more efficient shopping experience than can be achieved with Web 1.0.

The starter store can be used as a basis for your own customized store solution or you can use it as a learning tool.

Note: In the following chapters, we refer to the WebSphere Commerce Web 2.0 Starter Store as Madisons.

1.3.1 Introduction to the Madisons Store

Madisons uses common Web 2.0 features and capabilities such as Ajax and Dojo widgets to provide customers with an interactive and rich shopping experience. With a properly implemented Web 2.0 solution, this improves order conversion rates, because the Web 2.0 store is more customer-centric. The store can be used as a standalone store in the B2C business model or as an asset store in Extended Sites or Demand Chain business models.

The Madisons Store includes all of the features and capabilities of the Consumer Direct starter store. However, in this solution, the starter store features are enhanced from the Web 1.0 stores and provide a more interactive UI for customers. The shopping experience mimics using a Windows® desktop

application rather than the traditional static Web pages. The new capabilities of this Madisons Store are the Fast Finder page (which narrows product selection based on customer-selected attributes) and the convenient checkout (driving efficient checkout and decreased shopping cart abandonment).

The Madisons Store addresses many common Web customization issues, using CSS and DIV tags for page layout, design, and styling, instead of the current practice of overusing table elements throughout the page.

By taking advantage of Ajax and Dojo widget technologies, the Madisons Store delivers feature-rich and interactive Web applications that are compatible across platforms and browsers. Ajax technologies enable Web applications to retrieve data from the server asynchronously, while the browsing session remains intact. That is, refresh areas are present in pages for areas where content can be changed on the page, such as a shopping cart. Dojo widgets enable the users to interact dynamically with the pages (adding the ability to quickly view product information, rather than navigating to the specific product page, for example).

The store uses WebSphere Commerce services exclusively, while traditional data beans, that were used by the Web Sphere Commerce Web 1.0 Store, are used only in exceptional cases where corresponding services are not yet available (catalog, wishlists, content spots, and promotions, for example). For e-marketing spots, members, and orders, the store uses services to interact with the Commerce server.

The following list summarizes the capabilities of the Madisons Store:

- ▶ It is supported for both traditional Web 1.0 and current Web 2.0 shopping paradigms.
- ▶ It provides the ability to drag items into the shopping cart or a compare zone (for comparing same attributes among items).
- ▶ It contains e-marketing spots with an automatic scroll of items, products, categories, and merchandising associations.
- ▶ It provides product detail pop-up windows when customers hover over items.
- ▶ It has a Fast Finder, which narrows product selection based on customer-selected attributes (such as brand or price range).
- ▶ It provides the ability to change product attributes directly on the shopping cart (such as the color of fabric).
- ▶ It contains enhanced error-handling with a Web 2.0 error display, where the field with the error is highlighted with a useful message beside it.
- ▶ It provides an easier and more streamlined checkout experience.
- ▶ It provides greater accessibility functionality, enabling those with disabilities to interact with Web content through the use of assistive technologies.

1.3.2 Simplifying customization

The Madisons Store is up-to-date with current Web storefront designs that include typical Web 2.0 capabilities. The store uses full CSS and DIV elements for page design, page layout, and styling. This makes it easy to make simple changes to the look and feel in quick demo scenarios.

The customer is able to make simple cosmetic changes to the store by modifying the CSS instead of digging through the HTML code hidden within JSTL, script and Java code inside Java Server Pages (JSP™), and JSP fragments.

Some additional approaches for simplifying customization have been followed in this Madisons Store:

- ▶ Keep code modular. Use snippets when it makes sense.
- ▶ Add lots of comments to the page.
- ▶ Add debug statements in the JSP.
- ▶ Strictly follow JavaScript guidelines.

1.4 Applying Dojo and Ajax to the Madisons Store

In section 1.3.1, “Introduction to the Madisons Store” on page 10, we discussed the Madisons Store’s use of a new programming model that includes the use of Dojo widgets and Ajax. Ajax and Dojo widget technologies provide interactive Web applications that are compatible across many platforms and browsers.

In this section, we discuss the architecture of the Madisons Store and how Ajax and Dojo widgets are applied.

1.4.1 Architectural overview

The Madisons Store uses WebSphere Commerce services when interacting with WebSphere Commerce Server. Figure 1-2 illustrates what happens when the Madisons Store interacts with WebSphere Commerce Server.

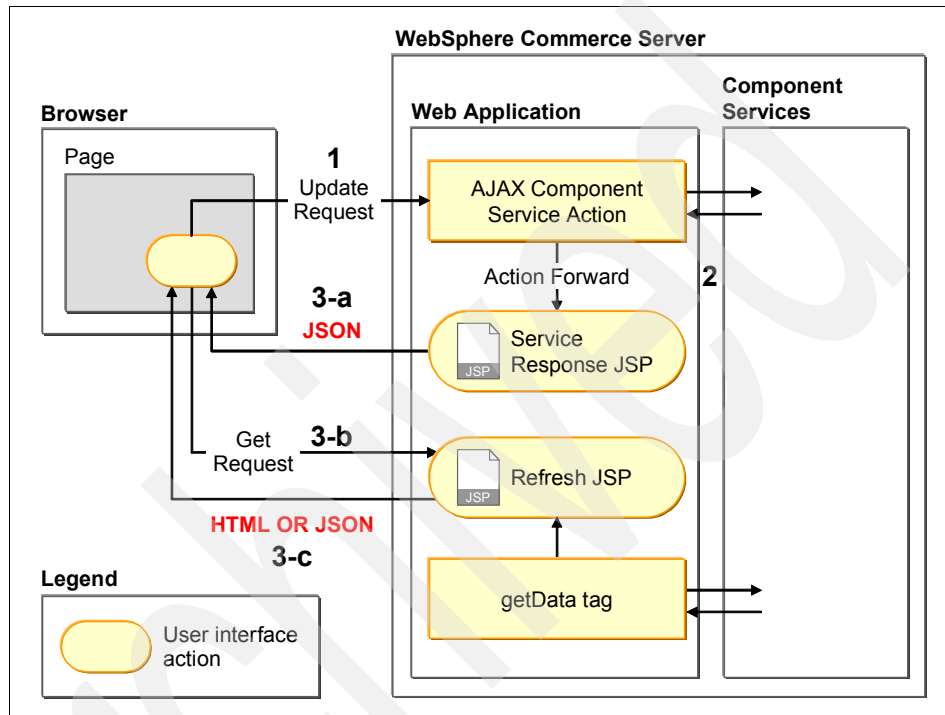


Figure 1-2 Interaction between Madisons Store and WebSphere Commerce server

The interaction shown in Figure 1-2 can be explained with the following process:

1. An Ajax update request using WebSphere Commerce services must be defined as a Struts action of type `AjaxComponentServiceAction`.
2. The system looks up what service to call by looking at the path to which this particular `AjaxComponentServiceAction` maps. Each service that is used as an `AjaxComponentServiceAction` must be registered in the `struts-config-ext.xml` file.

3. After the completion of the command, the Struts action forwards a JSP file that generates a JSON object containing either success or failure messages with all the contents in the response property. When the scenario is successful, the following events occur:
 - a. The client makes subsequent Ajax GET requests to the server to retrieve the data that should be updated on the Web page, as a result of the update request.
 - b. The JSP file uses the new wcf:getData tag to access information from the WebSphere Commerce database.
 - c. The JSP file builds the response as HTML or JSON code, which is then added to the page using JavaScript.

1.4.2 Common interaction

The common interaction in the Madisons Store works in the Publish/Subscribe model. Because the Madisons Store page will asynchronously display the up-to-date information of interaction with the server side without the whole page refreshed, some refresh area are defined to display the refreshed information. This is where the refresh area register themselves to listen to certain events. The events of interests are the ones that may cause its content to refresh.

Considering a shopping cart page that has several logical sections (such as shopping cart content, e-marketing spot content, mini shopping cart total in the header and a payment accordion that displays the total amount due), the UI makes use of the events system provided by Dojo to coordinate the common interaction of the store (Figure 1-3 on page 15).

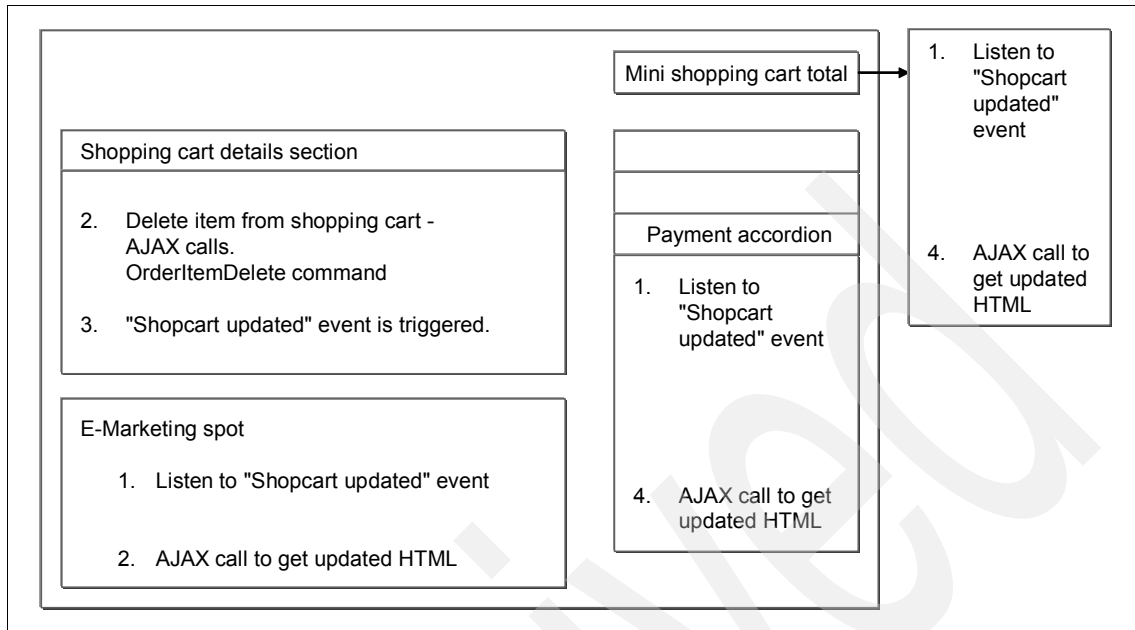


Figure 1-3 An example: Interaction in the Madisons Store

The interaction shown in Figure 1-3 can be explained with the following process:

1. At page initiation, all the page widgets or contents register themselves as interested on the "shopcart updated" event.
2. The shopper deletes an item from the shopping cart by dragging the item from the shopping cart contents into a trashcan icon.
This starts an Ajax update call to the deleteOrderItems service. After the logic in the server side executes, the JSP response will return a successful message.
3. The Ajax callback function triggers a "shopcart updated" event in step 3.
4. Each widget or page content that subscribes to listen to this event is notified. They can then execute their code to update their own contents.

This approach shows that widgets or sections within the page are modular and that they do not need to know about other parts in the page. Only the name of the events must be agreed upon within the widgets or code sections.

1.5 Madisons Store Feature Pack 5 features

The first version of Madisons Store was delivered in WebSphere Commerce V6.0 Feature Pack 2 as a reference application. In WebSphere Commerce V6 Feature Pack 5, the new Madisons Store is delivered in the base product.

The following features of the Madisons Store are new in Commerce V6 Feature Pack 5.

- ▶ Addresses total cost of implementation
- ▶ Support of traditional Web 1.0 and current Web 2.0 shopping paradigms, with an option for the store administrator to configure these paradigms using the Change Flow in Accelerator
- ▶ Accessibility improvements

WebSphere Commerce Web 2.0 Store features and benefits

This chapter details some of the Web 2.0 Store solution's benefits for the client, describes some of the most important Web 2.0 features of the new Web 2.0 Store solution of WebSphere Commerce, and introduces the Model View Controller (MVC) design pattern for Web 2.0. At the end of the chapter, we provide guidance to developers who are building and customizing Ajax-based applications, in preparation for the design and implementation guidance in Chapter 3, "Design and implement your Web 2.0 Store" on page 93.

2.1 Interactivity for user and benefits to clients

This chapter details ergonomic features for making users more comfortable with the Web 2.0 Store application interface.

2.1.1 Rich shopping experience

Web 2.0 is all about making your e-commerce Web site more appealing, interactive, and easy for your customers to use intuitively. The richer and easier the shopping experience, the more likely your customer is to make a purchase and return for repeat purchases. Making your products easy for customers to find is imperative, and Web 2.0 e-commerce technologies help with that.

The following features of the Web 2.0 Store improve conversion rates:

- ▶ Ajax technologies deliver Web pages, allowing richer product details to be downloaded quickly
- ▶ Visually richer pages that display products options, allowing customer interactivity without reloading or refreshing the Web page
- ▶ Dragging the products to a compare zone (Figure 2-1 on page 19) allows shoppers to look at more products side by side, and make a decision about purchase




Compare Products			
You can only compare up to 4 products			
	✖ Remove	✖ Remove	✖ Remove
Product Image			
	Add to Cart	Add to Cart	Add to Cart
Description	Enzi Deluxe Espresso Machine, Red	Kitchen's Best Digital 12 cup Coffee Maker, Blue	AromaStar 4 cup Steam Espresso & Cappuccino Machine
Price	\$179.99	\$99.99	\$69.99
Brand	Enzi	Kitchen's Best	AromaStar
Timer		✓	
Espresso	✓		✓
Auto-off		✓	✓

Figure 2-1 Compare Products page

- ▶ E-mailing products in a wishlist allows shoppers to send products to friends, who can make a purchase
- ▶ Fast Finder, which narrows product selection based on customer-selected attributes (such as brand or price range)
- ▶ Tagging, enhanced e-mail-a-friend, RSS feeds let you keep your customers informed about new and changing products, while getting your products in front of a wider audience
- ▶ Dynamic delivery of personalized page templates, layouts, and functionality based on customer preferences.

Madisons Store features that streamline the checkout process

The following features found in the Madisons Store streamline the checkout process:

- ▶ If Ajax checkout is enabled, shoppers can update their shopping cart without refreshing the whole page, only the changed areas
- ▶ Clicking **Add to cart** next to the recommended products on the shopping cart page adds the product to the shopping cart. The shopping cart area is then refreshed. The mini shopping cart details are updated with the current number of items in the cart, and the total price.
- ▶ Clicking **Remove** below the item removes it from the cart. The shopping cart area is refreshed to show the updated list of items and the updated price. The mini shopping cart details are updated with the current number of items in the cart, and the total price.
- ▶ Update the quantity of one of the items and the shopping cart is updated. The shopping cart area is refreshed to show the updated list of items and the updated price. The mini shopping cart details are updated with the current number of items in the cart, and the total price.
- ▶ Clicking **Add to wish list** below the item moves it to the wishlist. The shopping cart and mini shopping cart details are updated with the current number of items in the cart, and the total price.

Figure 2-2 shows the Shopping Cart Page after adding a product to the cart.

FurnitureTablewareKitchenwareApparel

Cart: 1 item(s) \$115.19

Continue Shopping


Cooking Oils

Frying Pans

Pots

Accessories

Coffee Makers

PRODUCT	AVAILABILITY	QTY	EACH	TOTAL
 Modern Occasional Table Remove Move to Wish List	In-Stock	<input type="text" value="1"/>	\$159.99	\$159.99

Save 20% on Furniture! (\$32.00)

Promotion code: [Apply](#)

New Customer & Guests

Checkout without signing in

You can make your purchases from Madisons without signing in

You will have the option to register and save in the checkout steps.

Returning Customers

Sign in for quick checkout

Username:

Password:

[Forgot your password?](#)

[Continue Checkout](#)


[Sign in & Checkout](#)

Customer Support


Need some help with your account?

Recommendations


You may also like:



Olive Oil Gift Set
~~\$29.95~~
\$24.99
[Add to Cart](#)



5-Piece Kitchen Utensil Set
~~\$9.95~~
\$8.99
[Add to Cart](#)



5-Piece Everyday Silverware Set
~~\$19.99~~
\$14.99
[Add to Cart](#)

Figure 2-2 Shopping Cart page with one item

Chapter 2. WebSphere Commerce Web 2.0 Store features and benefits 21

Figure 2-3 shows the Shopping Cart Page after adding another product to the cart. Only the shopping cart area is refreshed, and the mini shopping cart is updated with the new product.

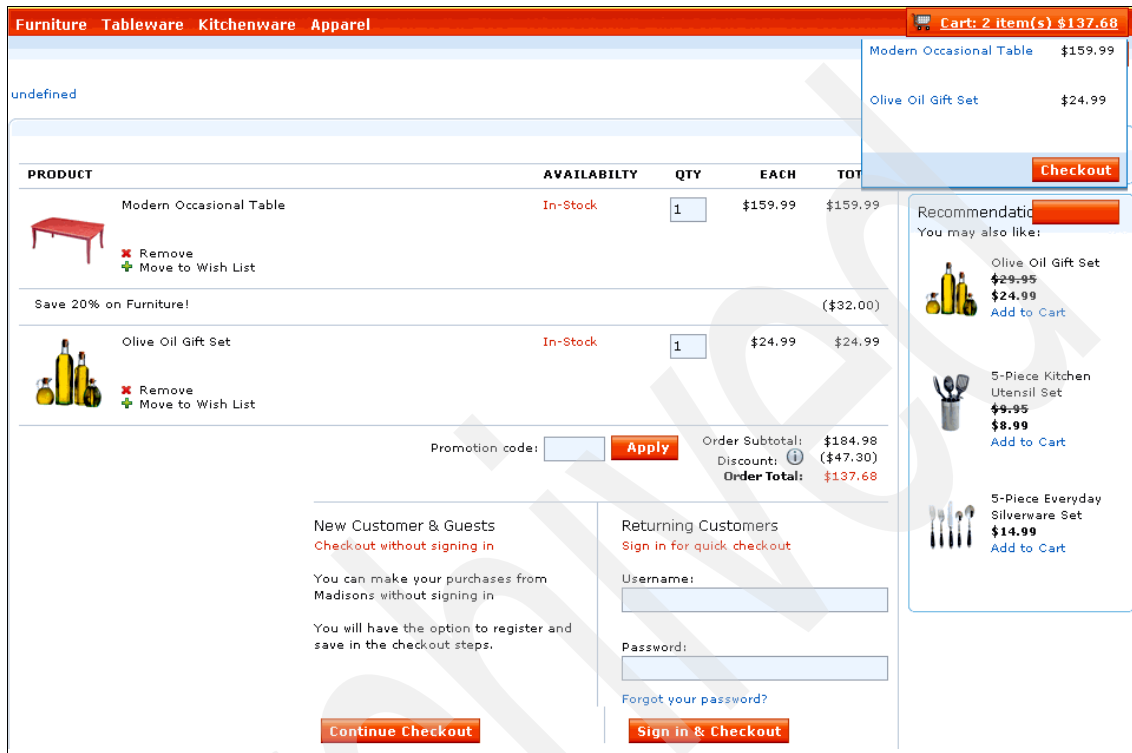


Figure 2-3 Shopping Cart page with two items



Web 2.0 Store features that offer a smooth e-commerce checkout process

Making the order and payment process as quick and simple as possible will dramatically improve conversion rates and boost sales. Web 2.0 e-commerce tools enhance the buying experience by making the process quick and simple for customers.

The following features offer a smooth checkout process:

- Dynamically linking to customer accounts with the Quick Checkout profile feature speeds up the checkout process, reduces the need for the customer to re-enter data, and allows quick selection from multiple delivery addresses

Shipping Address: Default_Shipping_10001		Shipping Method: International Priority	
Jhon Smith street 23 city province Aruba 1234124 23452353 jhon.smith@yahoo.com <input type="button" value="Edit"/>		<input checked="" type="checkbox"/> Ship as Complete <input type="checkbox"/> Advanced Shipping Options	

PRODUCT	AVAILABILITY	QTY	EACH	TOTAL
 Sharpson Deluxe 10 cup Digital Coffee Maker <input type="button" value="Remove"/> <input type="button" value="Move to Wish List"/>	In-Stock	1	\$39.99	\$39.99
 Sharpson 12 cup Programmable Coffee Maker, Green <input type="button" value="Remove"/> <input type="button" value="Move to Wish List"/>	In-Stock	1	\$69.99	\$69.99

Promotion code:

Order Subtotal:	\$109.98
Discount:	(\$11.00)
Tax:	\$0.00
Shipping:	\$0.00
Order Total:	\$98.98

Billing Information

If you want to use multiple payment methods select: 1 Payment method(s)

Billing Address: Default_Billing_10001		Billing Method: VISA Credit Card	
Jhon Smith street 23 city province Aruba 1234124 23452353 jhon.smith@yahoo.com <input type="button" value="Edit"/>		Card number: *****1111 CVV2 Number: <input type="text"/> What is CVV2? Month: 11 Year: 2010 Amount: 98.98	

Move on to your order summary.

Figure 2-4 Quick Checkout page

- Fewer page refreshes reduce the chance of data and customers being lost in transit
- Repeat purchases can be completed rapidly, often with just one click, while retaining the ability to change delivery and payment details.

2.1.2 Accessibility

In this chapter, we detail how RIA's technology addresses accessibility.

Definition of accessibility

Web accessibility defines how to make Web content usable by people with disabilities. People with some types of disabilities use assistive technology (AT) to interact with content. AT transforms the presentation of content into a format more suitable to the user, and allows the user to interact in ways different than the author designed. To accomplish this, the AT must understand the semantics of the content. Semantics are the knowledge of roles, states, and properties, as a person would understand them, that apply to elements in the content.

Accessibility problems

AT depends on extracting semantics from the document to represent it to the user. The HTML specification defines a limited number of semantic elements and attributes (for example, data tables within the page, or check boxes and radio buttons with a Web form). Web authors who want to create richer interfaces generally use generic HTML elements (`<div>` and ``) to build custom controls. These custom controls tend to have poor accessibility, because an AT cannot determine how to represent the generic HTML elements to the user. Unfortunately, HTML and other languages does not provide adequate markup to support accessible dynamic content.

Aspects of traditional HTML that make accessible support of dynamic content difficult are as follows:

- ▶ Accessibility relies on abstracting semantics from both content and presentation information. Extracting semantic cues from current HTML content is typically unreliable. Today, semantics are limited to tag element names.
- ▶ HTML allows content to be repurposed for presentation formatting without providing a way to convey semantic information. An example of this is using tables to format content instead of style sheets, or using `<div>` and CSS to define a custom widget.
- ▶ When combined with script and CSS, HTML can be repurposed to create dynamic custom components without providing a means to convey semantic information to native accessibility architectures designed to support dynamic GUI content.
- ▶ HTML lacks the ability to attach meaningful metadata about document structure.
- ▶ HTML elements commonly used for repurposing produce custom components that are not keyboard accessible.

Also, authors of JavaScript generated content do not want to limit themselves to using standard tag elements that define actual UI element such as tables, ordered lists, and so forth. Rather, they make extensive use of tag elements (such as <div>) in which they dynamically apply a UI through the use of style sheets and dynamic content changes. HTML <div> tags provide no semantic information.

Therefore, the developer of a Web application does not have the ability to provide the appropriate accessibility information in the markup to support the accessibility application programming interfaces (APIs) on the target platform.

As most of the Web sites today are rich Internet application (RIA) applications and contains JavaScript, they dramatically affect the ability for persons with disabilities to access Web content. New RIAs render custom widgets, modeling rich desktop components to perform UI updates without having to reload the entire page, much like a graphical user interface (GUI). Legacy GUI accessibility frameworks address these issues through a comprehensive accessibility API and infrastructure to foster interoperability with assistive technologies. These APIs constitute a contract between applications and assistive technologies (such as screen readers) to enable them to access rich dynamic content with the appropriate semantics needed to produce a usable alternative. No such contract exists between modern RIAs and assistive technologies, thus creating an accessibility gap for persons with disabilities.

Solving the accessibility problem

The Accessible Rich Internet Applications (ARIA) specification, given by the W3C - Web Accessibility Initiative's (WAI) Protocols and Formats working group (PFWG), tries to address these accessibility problems. The ARIA specification addresses the accessibility deficiencies in today's markup related to enabling RIAs by providing an extension to XHTML1.1, and a technique to support W3C ARIA in HTML 4.01. For this, ARIA defines two specifications:

- ▶ **ARIA States and Properties**

The ARIA States and Properties specification defines changeable states and properties of elements.

- ▶ **ARIA Roles**

The ARIA Roles specification helps identify element types that do not change with time or user actions.

For example, RIA developers can create a tree control in HTML using CSS and JavaScript even though HTML lacks a semantic element for that. A different element must be used, possibly a list element with display instructions, to make it look and behave like a tree control. AT, however, must present the element in a different modality and the display instructions may not be applicable. The AT will

present it as a list, which has display and interaction behaviors different than a tree control, and the user may be unsuccessful at understanding and operating the control.

The ARIA Roles specification provides a way for an author to provide proper semantics on custom widgets (like tree control, in the paragraph above) so that these custom widgets are accessible, usable, and interoperable with assistive technologies. This specification identifies the types of widgets and structures that are recognized by accessibility products by providing an ontology of corresponding roles that can be attached to content. This allows elements with a given role to be understood as a particular widget or structural type regardless of any semantics inherited from the implementing technology.

The ARIA States and Properties specification is used to declare important properties of an element that affect and describe interaction. These properties enable the user agent or operating system to handle the element properly, even when these properties are altered dynamically by scripts.

Note: More information about ARIA, ARIA-Roles, ARIA States and Properties, and ARIA best practices can be found at the following Web page:

<http://esw.w3.org/topic/PF/ARIA>

Accessibility for Dojo widgets

The widget set of the Dojo 1.0 release has been made accessible using ARIA. It provides keyboard support (widgets work with both keyboard and mouse) for all the core widgets. The ARIA States and Properties specification is applied to the Dojo widgets to allow for full access using screen readers. Developers using the Dojo 1.0 widgets do not need to be aware of the details of ARIA support. Developers creating new widgets or customizing existing widgets need to be aware of the dijit APIs for supporting ARIA. These APIs are found in `dijit.util.wai.js`, in the `dijit` branch of the Dojo source tree.

Accessibility implementation

W3C defines an extensible framework for Web authors to declare the role of each element on the page. The WAI Working Group is using this extensible framework to create taxonomy of roles [WAIRole] and states [WAISState] to describe common UI controls that are not represented in XHTML. Web authors can declare these roles and states on any element within an XHTML document, to indicate that the element represents a custom interface control. User agents can retrieve this accessibility metadata from the DOM and expose it to assistive technologies through the accessibility architecture of the underlying operating system. However, the methods described in [ROLEMOD], [WAIRole], and [WAISState] cannot be used directly in HTML4 documents, because they rely on

namespace features of XHTML that HTML documents do not support. The PFWG has come up with a technique to define the roles and states in the HTML4 documents. It allows the role and state information to be embedded into the class attribute so it becomes part of the DOM when the document is loaded.

Note: More information about ARIA roles and properties for commonly used elements like buttons, menus, combo boxes, and select boxes can be found at the following Web page:

<http://esw.w3.org/topic/PF/ARIA/BestPractices/SimpleExamples>

Accessibility for RefreshArea widget

One of the important things that should be addressed during development is the RefreshArea widget. RefreshArea's should be treated as *Live Regions*. Live Regions, according to ARIA specification, are parts of Web page that the Web author expects to change dynamically. Live Regions enable assistive technologies, such as screen readers, to be informed of updates without losing the users' place in the content. Live Region settings provide hints to assistive technologies about how to process updates. Live Regions allow text to be spoken to the user without getting misinterpreted. The ARIA State and Properties module defines a set of attributes which are related to Live Regions.

More information can be found on the following Web pages:

http://developer.mozilla.org/en/docs/AJAX:WAI_ARIA_Live_Regions

<http://esw.w3.org/topic/PF/ARIA/BestPractices/LiveRegion>

<http://juicystudio.com/article/wai-aria-live-regions.php>

2.1.3 Reducing the total cost of implementation

The new Madisons Store is up to date with current Web storefront designs that include Web 2.0 capabilities. It uses CSS for styling and DIV elements and layers for designing, and for ease of implementation and change. You can easily make changes to the CSS files instead of searching through the HTML code, which is hidden in JSTL files, or through Java code inside Java Server Pages (JSP) and JSP fragments.

The following list details a few design principles of Web 2.0 that can reduce total cost of implementation (TCI):

- Keep code modular. Use snippets when it makes sense.

Not all pages need to be broken down into JSP snippets. Use snippets only when it is necessary, when there is reusable code fragments, or when it is a piece of code that shows a particular feature.

Keep the code modular by having the bean initialization next to where it is used. Do not have a section at the top of the page to do all the initialization and then use them in different places in the page. Try to keep common code in blocks to make it much easier for developers to move code around to a different page or to delete functions that are not required.

► Add lots of comments to the page

Each JSP page must explain what it does, what it displays, and any other important information. For reusable snippets, clearly document the usage and the accepted parameters. At different blocks of code, add some comments to explain what is being done.

Example 2-1 shows how comments should be added in a main JSP page.

Example 2-1 JSP comments

```
<%--
* A brief summary description of the page and what is being
displayed.
* List all JSP and JSPF that are being included by the page.
--%>
Sample for a JSP snippet or fragment:
<%--
* A brief summary description of the snippet or fragment.
* List all required and optional input parameters, clearly
explaining the default
    values for the optional parameters.
* (Optional) Sample usage of the snippet.
* List all the JavaScript files that must be included by the main
page if this snippet
    or fragment is used.
--%>
```

► Add debug statements in the JSP

Use the Dojo debug utility to create traces in JavaScript code. For any other tracing requirement it is suggested to use the server side utility (Java Logging API).

► JavaScript guidelines

Every JavaScript function and global variable must be defined within some namespace so that it does not collide with other code. The convention is to create a namespace for each page by creating a new object.

Example 2-2 on page 29 shows some namespace for a JavaScript function.

Example 2-2 JavaScript function namespace

```
// first, declare the namespace if it does not already exist
if (FF == null || typeof(FF) != "object") { var FF = new Object();}
// all the functions in the FF namespace will go in this block
FF = {
  filterProducts: function (oSelf, oOther) {
    // contents of the filterProducts function
  },
  createResults: function () {
    // contents of the createResults function
  },
  thirdFunction: function () {
    // function contents
  }
} // end of the FF namespace
```

- ▶ Make all JavaScript functions independent of JSTL so that they can be moved to separate JavaScript-only files. If required, accept parameters or objects with the values required. Declare a constructor for the object that is created.
- ▶ All JavaScript functions must have Java Doc explaining what its function is, with input and output parameters. List any dependant forms or input values in forms that it expects and uses. Also, clearly define any assumptions for which callers need to be made aware.

Example 2-3 shows how we can comment a JavaScript function.

Example 2-3 Java Doc

```
////////////////////////////////////
// A short description of what type of functions are defined in the
// page.
// State if dojo is a requirement when including this JavaScript file.
// List all dojo API (i.e. dojo.require) that needs to be added in the
// page including this file.
////////////////////////////////////
sampleFunction:function() {
  // summary      : initialize the Sample page
  // description   : Executed onLoad of Sample page. This
function
  //sends the    addressId/shipModeId of the first sample block in
this
  //page to the quick cart, so that when items are moved to quick
cart,
```

```
//they are added with this info by default. Then define the drop
//targets. Each shipment block will be on drop area. Also hides the
//default input date field that comes with Dojo's date picker
// assumptions      : Any assumptions made.
// dojo API         : Dojo API's used.
// Params           : Input Params
// returns          : returned value
}
```

2.2 Model View Controller design pattern in Web 2.0

This chapter will make an overview of the Model View Controller (MVC) design pattern and how this pattern applies to the Web 2.0 Store solution.

2.2.1 Model View Controller design pattern

The MVC pattern can separate business logic from UIs, meaning the underlying business logic, or the application's presentation may be altered without affecting the other.

Most developers of Web-based applications are familiar with the J2EE/JEE Model 2 design model, which uses the MVC pattern to separate the data content from the presentation. Figure 2-5 on page 31 shows how MVC works.

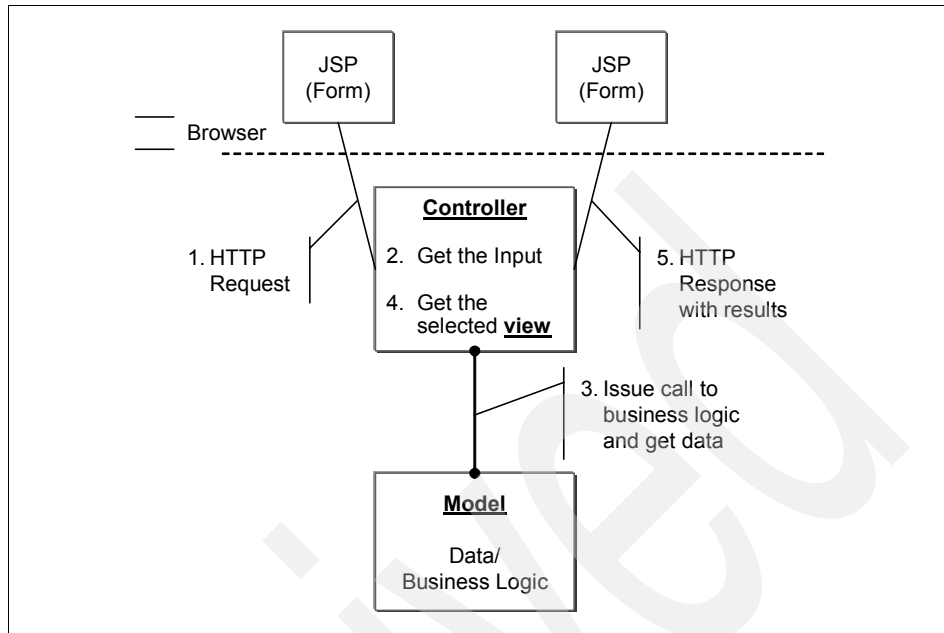


Figure 2-5 Model View Controller

MVC framework with Struts

Struts is a open source framework for building Web applications according to the MVC principles. It provides the following components to develop applications using MVC. See Figure 2-6 on page 32 for a graphic representation of these components.

► Model

Struts does not provide model classes. The business logic must be provided by the Web application developer using Enterprise JavaBeans™ (EJBs).

► View

Struts provides action forms to create form beans that are used to pass data between the controller and the view. In addition, Struts custom tag libraries assist developers in creating interactive form-based applications using JSPs. An application resource file holds text constants and error messages that are used in JSPs.

► Controller

Struts provide an action servlet (controller servlet) that populates action forms from JSP input fields, and then calls an action class where the developer provides the logic to interface with the model.

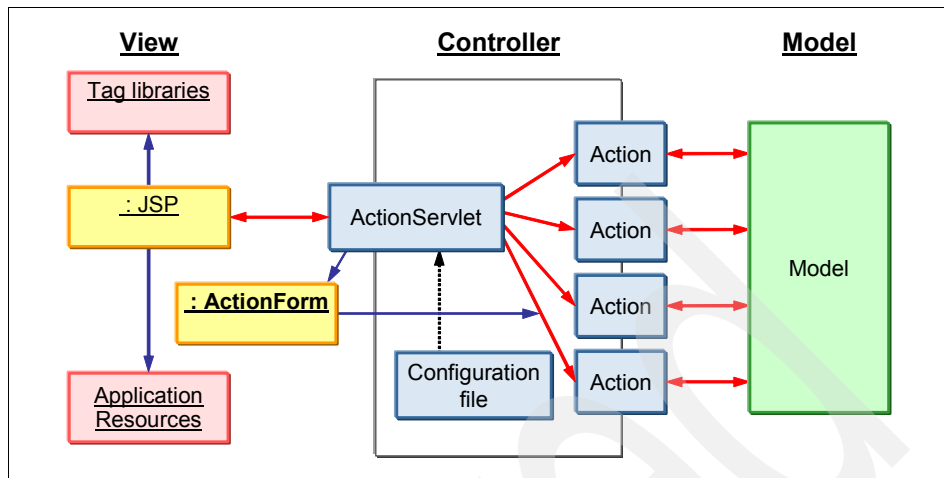


Figure 2-6 Struts Components in the MVC architecture

A typical Struts Web application is made up of the following components:

- ▶ A single servlet (`org.apache.struts.action.ActionServlet`), which uses an XML configuration file. The action servlet invokes actions based on the JSP's action specification, and it invokes JSPs based on the actions forwarding information.
- ▶ Multiple JSPs that provide the user view. Struts tag libraries make JSP coding easier.
- ▶ An application resources file, which holds text constants and error messages and makes internationalization easy.
- ▶ Multiple action classes (extending `org.apache.struts.action.Action`) to interface to the model.
- ▶ Multiple action forms (extending `org.apache.struts.action.ActionForm`) to hold the data from the JSPs. The action forms are initialized by the action servlet and passed to the action classes. Action forms can be used to validate the data entered by users.

2.2.2 How MVC applies for Web 2.0

In the Web 2.0 model, 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 the client can then begin to take over the UI controlling logic. To further reduce unnecessary costly interaction with the server, the client can maintain a local data store for UI responsiveness. Certain UI functions will trigger persisting local data to the server.

The Web 2.0 Store solution uses RIA technologies such as Ajax and Dojo widgets to provide customers with an interactive and rich shopping experience.

Web 2.0 Store solution

WebSphere Commerce is based on Struts Framework. All browser requests are sent to the Struts Action Servlet, which forwards the request to the corresponding action based on URLs and the mappings from the configuration XML file. The Struts Action will perform a service and return a Struts Action Forward, which is used to render the new page to return to the browser.

The new component model requires that components define a set of services that accept Open Application Group Integration Specifications (OAGIS) Business Object Document (BOD) messages. This BOD messages architecture is passed to the component façades in the form of Java objects that are consistent with the structure of the request BOD message. The component façade methods will return OAGIS BOD responses. Web developers will need to access these component services through regular Web 1.0 page load requests, through Ajax style requests, as part of render requests, and on the server while a JSP is being rendered.

In WebSphere Commerce, the action services are handled by controller commands. This allows Struts actions to be configured using a new action implementation that will invoke the new component services. JSP authors will require component data to render the page. The Java Client API provides methods for Java programmers that allow them to retrieve static business objects with the required component data. JSP developers have access to tag libraries that allow them to call Get services methods and retrieve a list of business objects that match a specified XPath search expression. The populated business objects can be used by the JSP developers like regular data beans. To include Ajax-style requests in a JSP to render HTML pages for updating parts of a Web page, these requests have to call existing controller commands as well as new component services.

Figure 2-7 on page 34 illustrates an Ajax invocation pattern for an action that invokes a component service and then updates the page as required to reflect the new state.

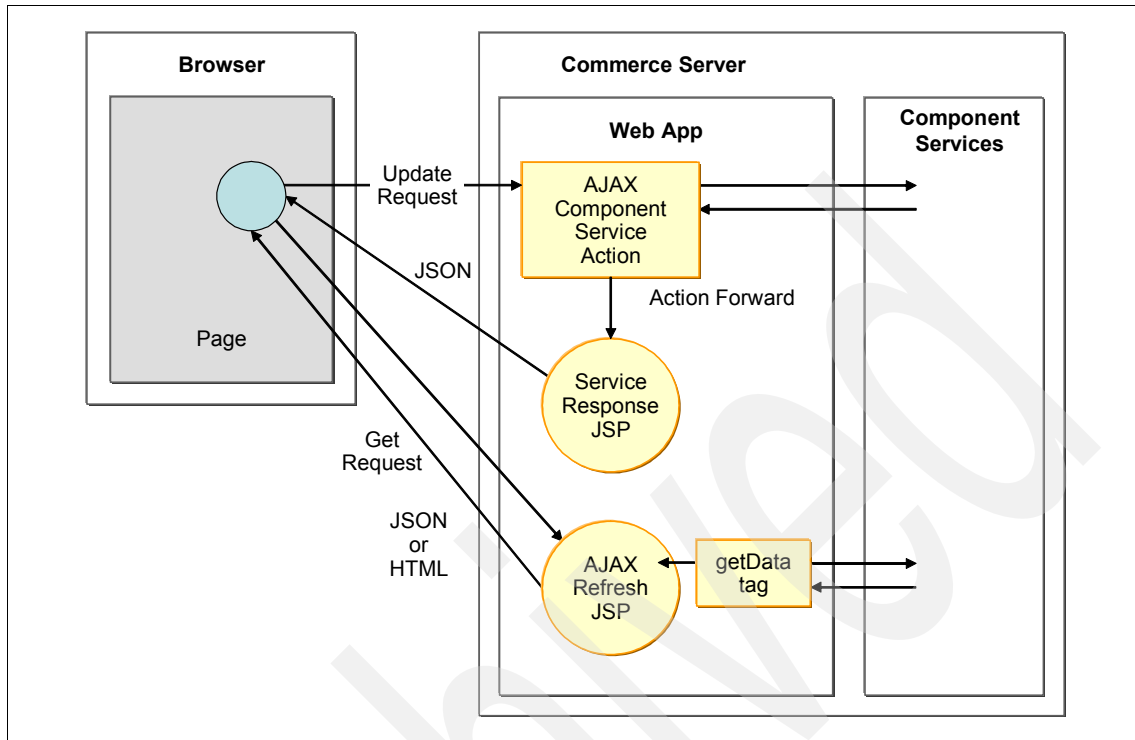


Figure 2-7 Ajax pattern using component service

An Ajax-style request requires an invocation of a service that will do some update of the request, followed by an update of some parts of the Web page to reflect the updated information. For example, a user drags an item to the shopping cart. This triggers an Ajax request to add the item to the shopping cart. When the item is added to the shopping cart, parts of the page will need to be refreshed to reflect the new state of the shopping cart.

The updated request will invoke a service that will perform the request. If the new component services are being used, then the request uses a new Ajax component service implementation that invokes the component service. The Ajax component service action is configured to invoke a Web-friendly method on the component client façade that will accept a Map of name value pairs. The method will construct the BOD from the Map and invoke the appropriate component service. When the service completes, the method will convert the response BOD into a response Map and return the response Map to the action implementation. The action implementation will return an Action Forward that invokes a JSP responsible for creating the service response. The service response is a JSON object that contains all of the values found in the response Map. This response can then be interpreted by the JavaScript response handler

on the page that submitted the request. The response will contain standard name/value pairs that indicate if the request was successful or not. In the case of an unsuccessful request, a message is displayed to the user and the page is left otherwise unaffected. In the case of a successful update request, parts of the page may need to be refreshed to reflect the updated state.

When the update completes successfully, the affected parts of the page will launch Ajax requests that allow the page fragments to be refreshed. The Get Ajax request calls the Struts Servlet which will delegate the request to a JSP. The JSP is able to retrieve the appropriate data from the component service using the `wcf:getData` tag and render either HTML or JSON. The response format should be in the form that is easiest for the JavaScript code that is updating the page with the new data. Experience shows that we want to avoid delegating the complicated logic of parsing and formatting the new data to JavaScript running in the browser. It is much more efficient to handle this on the server side.

Figure 2-8 illustrates the typical Ajax invocation pattern for an action that invokes a controller command to perform a service, and then updates the page as required to reflect the outcome of the service.

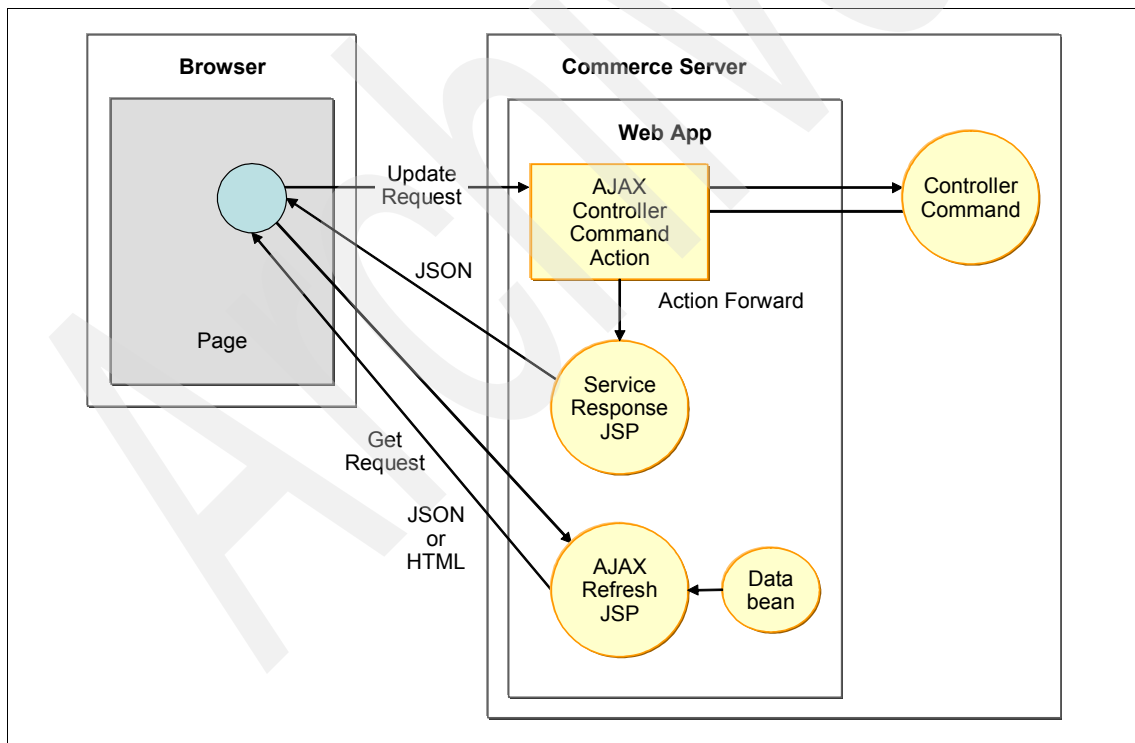


Figure 2-8 Ajax Pattern using controller commands

The invocation pattern is the same as the pattern used when component services are used to perform the service.

When to use Ajax invocation pattern

The Ajax invocation pattern was derived from two design principles:

- ▶ To ensure that the well-defined component services can be effectively used by the Web developer

Component developers are required to provide Web-friendly methods that can be used to invoke the services by passing name/value pairs rather than constructing the OAGIS BOD message.

- ▶ To limit the complexity of the JavaScript required to update the page

Experience with JavaScript in the tools framework has taught us that JavaScript does not scale when it is required to format and parse large amounts of data. By performing the complicated parsing and formatting using JSP technology (designed for that purpose), new page fragments can be returned to the browser in a preformatted response that is ready to be inserted into the page with a minimal amount of JavaScript.

In addition to the Ajax considerations, care has been taken to ensure that exploiting these new component services does not prevent Web developers from using the existing Struts actions and data beans. It is conceivable that a Web application may make use of both the new techniques for invoking new component services and the existing techniques for invoking Struts controller commands and data beans. It is possible that a single JSP may use the new get data tag library to get data from a new component service, and also use a data bean to retrieve from an existing Commerce component.

If the Web application is being rendered outside of the Commerce application, the Commerce Struts actions and data beans will not be available. For example, if the Web applications are part of a portal application, the developer will only be able to exploit Commerce through the component services.

Note: For implementation of the Ajax invocation pattern in the Web 2.0 Store solution, see “Scenario 3: Updating business objects in WebSphere Commerce using Ajax, and returning all relevant update information through JSON” on page 133.

2.3 Web 2.0 Store (FEP2) features and benefits

The Madisons Store includes all of the features and capabilities of the consumer direct starter store. However, in this solution, the starter store features are enhanced from the Web 1.0 stores and provide a more interactive UI for customers. The shopping experience mimics using a Windows desktop application rather than the traditional static Web pages. The major attraction of Madisons are the Fast Finder page (which narrows product selection based on customer-selected attributes) and a single page checkout (which drives efficient checkout and decreased shopping cart abandonment).

The following list details features included in the Web 2.0 Store solution:

- ▶ The ability to drag items into the shopping cart, wishlist, and compare zone.
- ▶ e-marketing spots with automatic scroll of items, products, categories, and merchandising associations.
- ▶ Product detail pop-up windows when customers hover over items.
- ▶ A Fast Finder page which narrows product selection based on customer-selected attributes such as brand or price range.
- ▶ The ability to change product attributes (such as color or fabric changes) on the shopping cart.
- ▶ Single page checkout.
- ▶ WebSphere Commerce Feature Pack 2 Coremetrics analytics integrated into the RIA technologies.

As you deploy the Web 2.0 Store, follow the recommendations we describe in Chapter 4, “Testing and debugging your Web 2.0 Store” on page 195. The installation and deployment instructions result in Dojo making Ajax-style requests to get its source code at runtime for every page and every function used. Chapter 3, “Design and implement your Web 2.0 Store” on page 93, has information about how to create the custom Dojo source file for the application, so Dojo will not have to send any Ajax requests to retrieve source code. This significantly improves performance. Another important recommendation involves Dojo parsing. Follow those techniques in every new and modified page that you develop or change. Without those techniques, the client-side browser processing when a page is loaded is impacted.

2.3.1 Single store supporting Web 1.0 and Web 2.0

In Commerce V6, feature pack 2, we introduced a reference application store to show Web 2.0 shopping paradigms. The Consumer Direct is a traditional Web 1.0 store with full page refreshes. Although both stores were based on B2C business models, and both implemented the same set of basic functionality, they differed in the presentation and the interaction features provided to the shopper. Both these stores are merged into Madisons, making the selection much easier for the customer. This feature is implemented using the current Change Flow functionality available in WebSphere Commerce. The store administrator will have an option to select either a Web 1.0 or Web 2.0 path using Change Flow features at any point of time. These Change Flow options allow the store administrator to choose individual Web 2.0 features instead of an all-or-nothing scenario.

2.3.2 Madisons Store Change Flow function options

The Madisons Store has the following options in the Change Flow function from Commerce Accelerator. The store administrator can enable any of the features as required.

Registration page options

The following list details the Registration page options.

- ▶ Enable preferred currency selection
- ▶ Enable preferred language selection
- ▶ Collect age information during registration
- ▶ Collect gender information during registration
- ▶ Enable promotional E-mail option during registration
- ▶ Enable remember me selection

Personal information management page options

The following list details the Personal information management page options.

- ▶ Enable order status tracking
- ▶ Show shipment tracking URL
- ▶ Enable Ajax in the my account page (If enabled, all the tasks within the “my account” page is done in a single page using Ajax.)

Catalog options

The following list details the catalog options.

- ▶ Enable search in the store
- ▶ Enable quantity field selection
- ▶ Display product only (If enabled, only products are displayed in product display page. Otherwise, the product display page will contain products as well as items.)
- ▶ Enable wish list
- ▶ Enable interactive scrolling e-marketing spot
- ▶ Enable product comparison
- ▶ Enable drag and drop to compare

Order options

The following list details the order options.

- ▶ Enable quick order
- ▶ Display mini shopping cart on header
- ▶ Enable drag and drop to mini shopping cart
- ▶ Enable interactive mini shopping cart display on item added to cart (If enabled, the contents of the shopping cart are briefly shown when an item is added to the order.)
- ▶ Enable Ajax add to shopping cart (If enabled, clicking the **add to shopping cart** button is done using Ajax, so the shopper will stay in the current page after a successful add. Otherwise, the **add to shopping cart** button redirects the user to the shopping cart page.)

Checkout options

The following list details the checkout options.

- ▶ Enable quick checkout
- ▶ Enable multiple shipments
- ▶ Enable requested shipping date
- ▶ Enable shipping instructions
- ▶ Enable promotion code entry field
- ▶ Enable Ajax checkout (If enabled, all the tasks within the checkout page are done in a single page using Ajax. With Ajax checkout there are no page refreshes and the information in the checkout page is updated asynchronously.)

2.3.3 Accessibility issues

Web accessibility defines how to make Web content usable by people with disabilities. People with some types of disabilities use Assistive Technology (AT) to interact with content. AT can transform the presentation of content into a format more suitable to the user, and can allow the user to interact in ways different than the author designed.

The Madisons Store is coded with accessibility support. Implementation of this feature adheres to the Accessibility Rich Internet Applications (ARIA) specification given by the W3C Web Accessibility Initiative's (WAI) Protocols and Formats working group (PFWG), making the store fully accessible.

2.3.4 Browser backward and forward buttons

The Madisons Store makes use of the `dojo.undo.browser` API for handling backward and forward browser actions. The idea is to save some state objects that contain functions to handle backward and forward. It uses Dojo APIs to register these different state objects to the browser's history as required. When the browser's **back** button is pressed, the back function of the saved state is called (likewise when the forward button is clicked). Visit the following Web page for more details:

<http://manual.dojotoolkit.org/WikiHome/DojoDotBook/Book0>

2.3.5 Tooltips

In the User Registration form, when a shopper clicks on the **Submit** button and no valid data is filled in, a tooltip with fill-in details displays on the right side of the invalid field, as shown in Figure 2-9 on page 41.

Please Register Below

* required fields

Logon ID: The Logon Id field cannot be empty.

Password:

Verify password:

First name:

Last name:

Street address:

City:

Country/Region:

State/Province:

IP code/Postal code:

E-mail:

Phone number:

☐ Send me e-mails about store specials.

Preferred language:

Age: [Privacy Policy](#)

Gender:

☐ Remember Me

Figure 2-9 Tooltip for the input fields

2.3.6 The drag feature

Important: The store administrator must enable dragging to mini shops and compare zones. This is done through the Change Flow menu in the Commerce Accelerator.

Shoppers use the drag feature to move items or products to mini shopping cart accordion drop zones. Items are added to the cart and the page is updated to display the changes.

Note: Products that require customers to specify additional defining attributes (such as color, dimensions, or finish), cannot be dragged into the quick cart or the wishlist. They can only be dragged into a compare zone. You can, however, drag products not requiring additional customer preferences into all three zones.

Figure 2-10 shows an example of dragging an item into a compare zone.

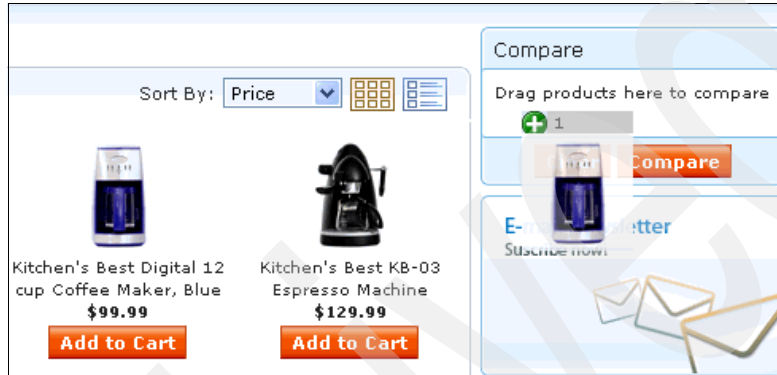


Figure 2-10 Dragging a product into a compare zone

The item can only be dropped in to a permitted area, denoted by the green + (plus sign) mark. If the item is dropped when the + (plus sign) mark is red or is not drop in a drop zone, the item is returned to its original position.

Shoppers can drag items displayed from the browser navigation flow, Fast Finder, or a marketing area to the following areas:

- ▶ Mini shop cart
- ▶ Compare zone
- ▶ Wishlist

2.3.7 Mini Shop cart

Important: The store administrator must enable the “Display minishop cart in the header” and “Drag & Drop to minishop” features from the Store/Change Flow menu in the Commerce Accelerator.

Shoppers can see the total shopping cart in the header of each page. To add a item to the mini shop cart, drag and drop the item to a position where the + (plus sign) mark turns green. Figure 2-11 shows an example of dragging an item into the mini shop cart.

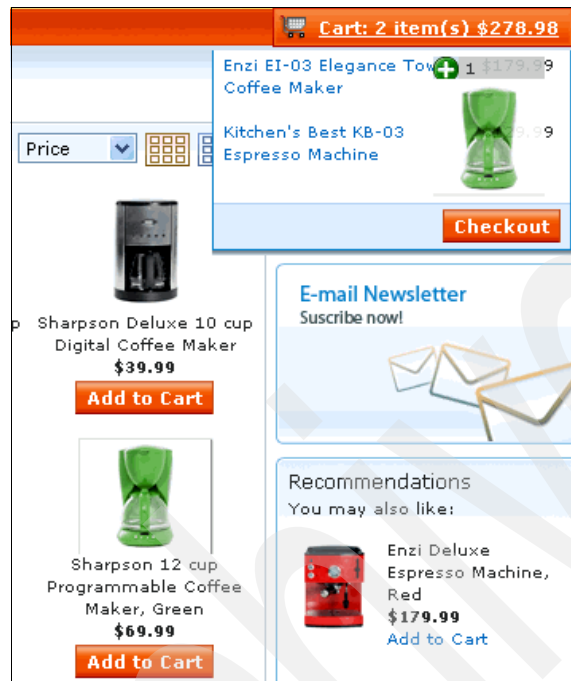


Figure 2-11 Dragging and dropping an item into the minshop cart

Dropping the item when the + (plus sign) mark is red, or in a non-droppable area, will return the item to its original position.

Hovering the mouse displays a tooltip with the name of the products in the cart and the corresponding price for each of them. See Figure 2-12 on page 44. Shoppers can access the Product Details page from this tooltip, or they can access the shopping cart page by clicking on the mini shop cart link.

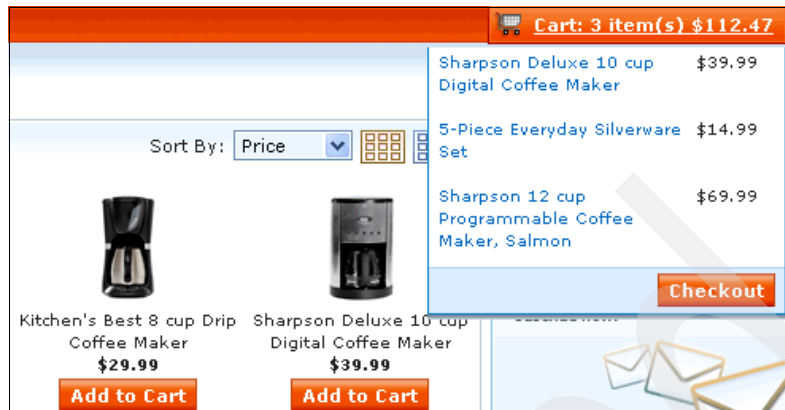


Figure 2-12 Mini Shop Cart

Clicking an item name will close the tooltip and display its detail page. If the shopper clicks the **Checkout** button, the tool tip will close and the shopping cart page is displayed. The shopping cart will contain items previously dragged into the mini shop cart, as shown in Figure 2-13.

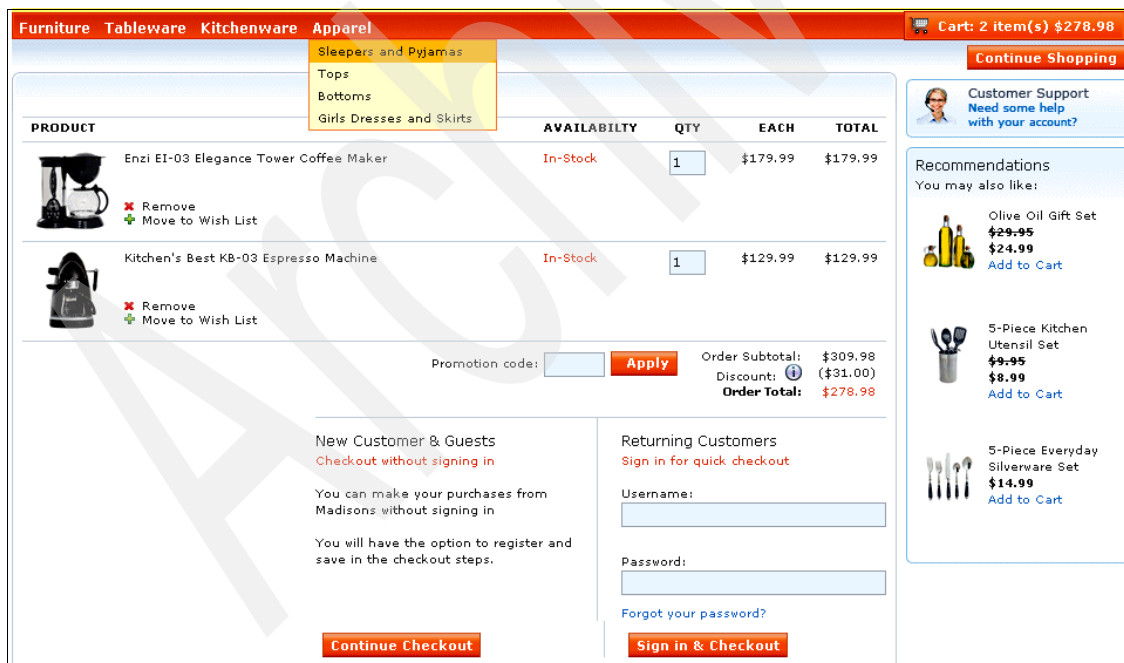


Figure 2-13 Shopping cart after dragging an item into the Mini Shop Cart

2.3.8 Quick Info panel

While hovering the mouse over a product image in the catalog navigation flow, compare zone, Fast Finder, or e-marketing spot, shoppers can click the **Quick Info** button below the item. This opens the Quick Info panel (Figure 2-14), a window with more details about the item. Shoppers can add the item to shopping cart, wishlist, or compare zone.

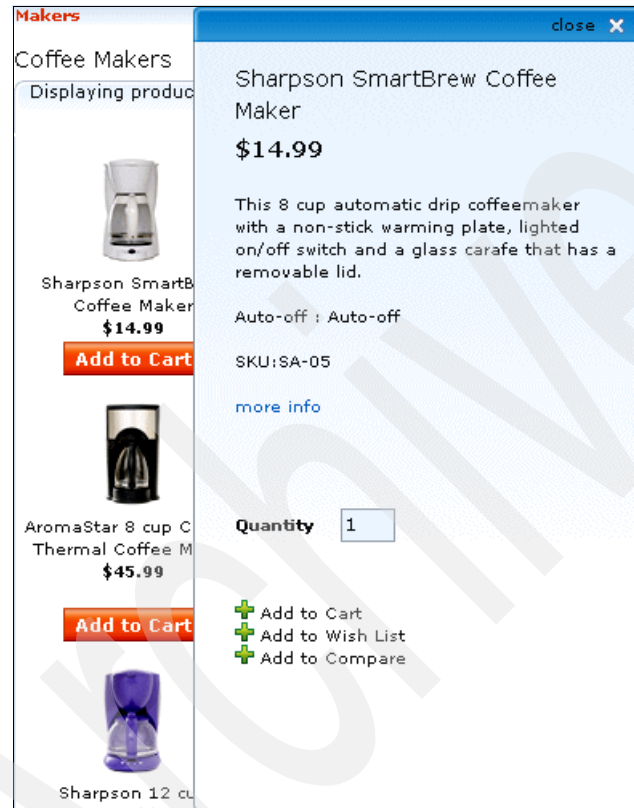


Figure 2-14 Quick Info

When the shopper clicks **close (x)**, the window will close. Moving the mouse over a different item will close the existing window and will open a new one for the current item. When the shopper moves the mouse over a different area on the page, the window closes, and all details about the item disappear.

When shoppers click the **more info** button, the window closes, and the product detail page displays details about that item.

2.3.9 Scrollability widget

Important: The store administrator must enable interactive scrolling. This is done in the Store/Change Flow menu in the Commerce Accelerator.

When enabled, the Scrollable widget is displayed on the home page of the site in an e-marketing spot, as shown in Figure 2-15.

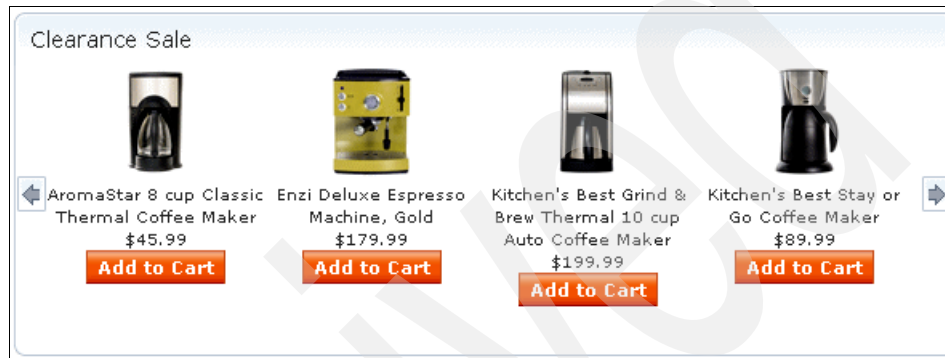


Figure 2-15 Scrollability widget

Hover the mouse over the e-marketing spot to stop the automatic scrolling. Click the right navigation icon and the next set of items to the right are displayed. Click the left navigation icon and the next set of items to the left are displayed. Hover over product image and the **Quick Info** button displays. This button opens a tooltip with more information about the item and options to add to shopping cart, wishlist and compare zone.

Shoppers can drag items from the scrollable area into the shopping cart, compare zone, or wishlist.

Shoppers can add an item that is displayed in this scrollable area to the shopping cart with the **Add to Cart** button.

2.3.10 Fast Finder

Shoppers have the ability to narrow the product list by using Fast Finder filtering options, which are displayed in the left side of the product list page. As the shopper makes selections, the result set is updated on the page.

Figure 2-16 on page 47 shows an example of a page with Fast Finder feature.

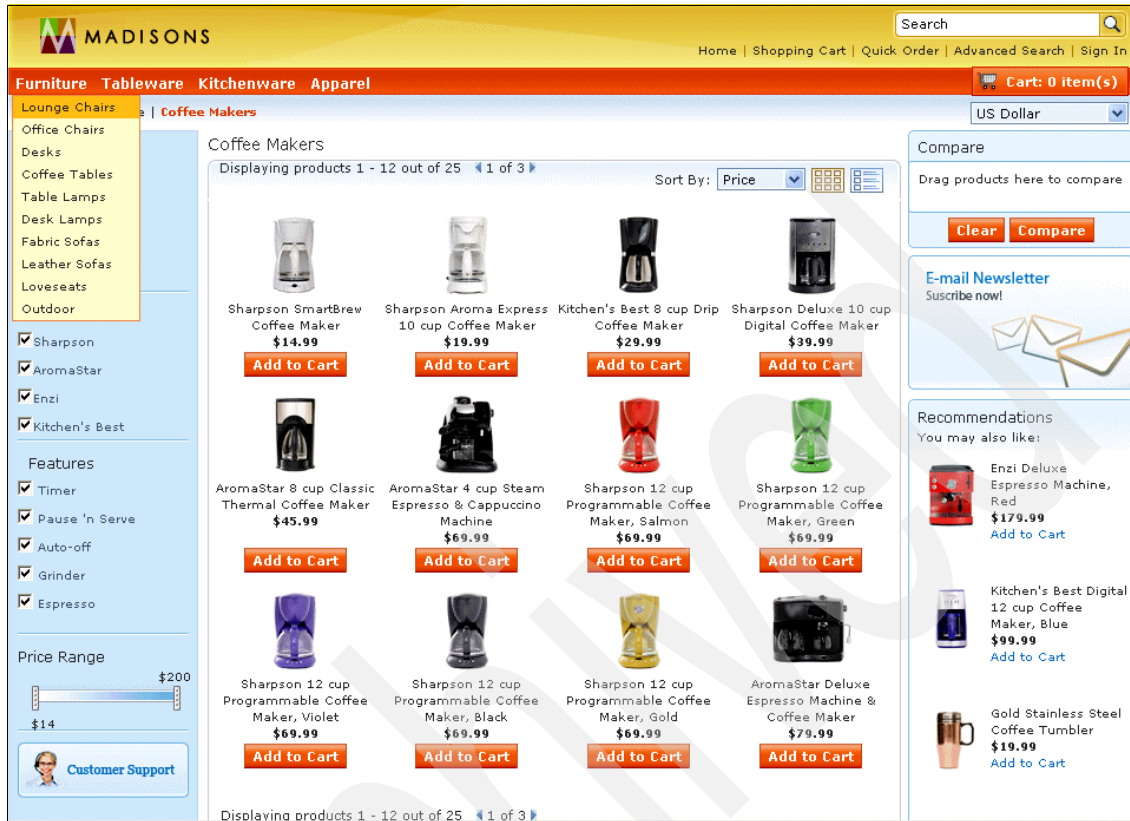


Figure 2-16 Fast Finder page

The left navigation column contains different attributes, which the shopper uses to narrow down the products list (for example, brand, features). It also contains a price range slider widget.

When the shopper selects an option, the resulting products narrow. An option is given to change the display details and the layout of the products displayed.

When the shopper moves the price range slider widget (shown in Figure 2-17 on page 48), the resulting products narrow to display only the items in the selected price range. An option to given to change the display details and the layout of the products displayed.

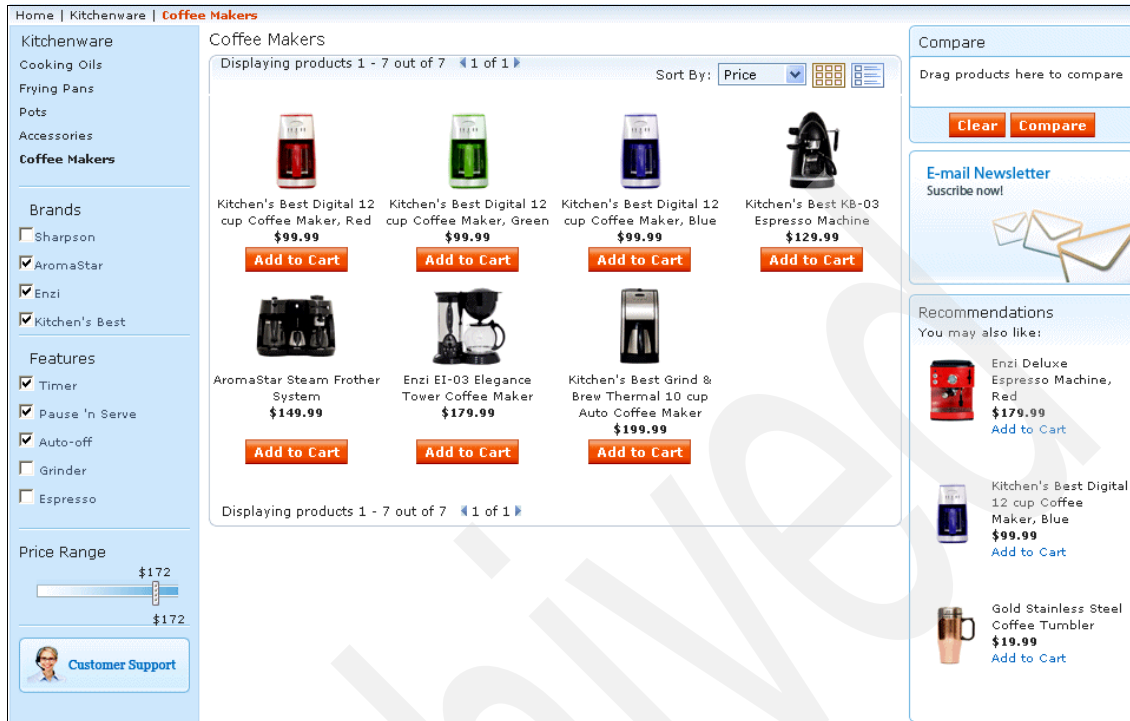


Figure 2-17 Fast Finder after narrowing by price range and features

The Fast Finder page is built using client side-based filtering, which means that when the page is loaded, the items are saved in a JavaScript local object and all filtering options are determined while rendering the items. Items in a category are sorted by brand and price are extracted using the CatEntrySearchListDataBean. When a price, brand, or other feature filter is used to narrow the products, JavaScript methods are called to generate the corresponding result set and change the display area in the page.

Optimization

The Fast Finder page uses full page caching from dynacache, which means that the page is cached to the Fast Finder at the first visit, and all subsequent hits on the same category display a cached page.

The Fast Finder page can be found in the following directory:

StoreName"/Madisons/AJAXUserInterface/ShoppingArea/CatalogSection/SearchSubsection/FastFinderDisplay.jsp

The Fast Finder page uses the `wc.widget.RangeSlider` and the `wc.widget.ProductQuickView` dojo widgets. The item pop-up window and the drag and drop features are properties of the `wc.widget.ProductQuickView` widget, which means the Fast Finder page does not need to code those properties.

2.4 Web 2.0 application concepts

This section details some of the basic concepts of designing a Web 2.0 application. It describes how can to include the Dojo toolkit in your Web application and what components you might use.

Web 2.0 refers to a new generation of the World Wide Web, which lets people collaborate and share information online. It gives the user a perspective that is closer to desktop applications than static online Web applications. Web 2.0 is a platform that incorporates the principles and practices shown in Figure 2-18.

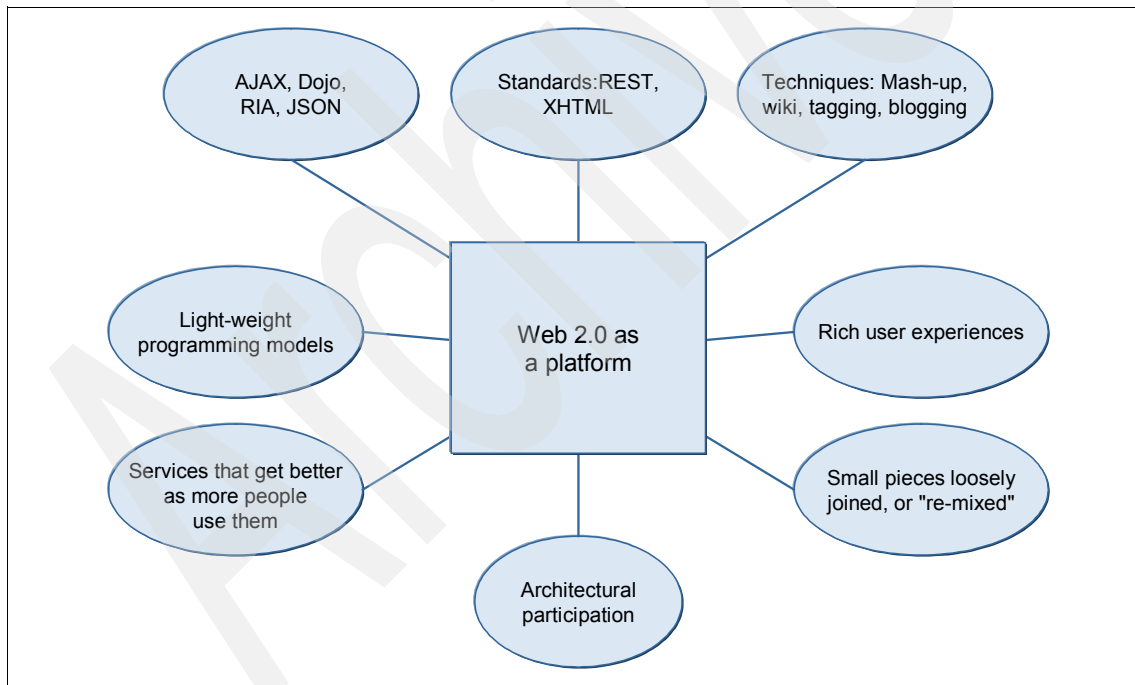


Figure 2-18 Web 2.0 platform

The concepts behind a Web 2.0 application are discussed in 1.1, “Introduction to Web 2.0” on page 2 and about the technologies used by developers of Web 2.0 applications. See below references from 1.1.2, “Technologies behind Web 2.0” on page 4, and also 1.1.4, “Rich Internet applications” on page 8

- ▶ “Ajax” on page 4
- ▶ “Atom feeds and JSON” on page 5
- ▶ “Dojo” on page 6

2.4.1 Overview of the Dojo toolkit

The Dojo toolkit is a multiplatform, open source JavaScript toolkit backed by IBM, Sun Microsystems, SitePen, AOL, and other companies. It is not tied to any specific server-side toolkit, so it can be used in conjunction with PHP, Java Servlet, Java Server Pages, active server pages (ASP), and even with mobile devices.

Dojo-related Web pages: Check the Dojo Web site for the latest information regarding the Dojo toolkit:

<http://www.dojotoolkit.org>

This chapter describes the main concepts of Dojo. It does not provide an API description of the toolkit. The Dojo toolkit APIs can be found at the following Web page:

<http://api.dojotoolkit.org>

The Dojo toolkit has a modular design, consisting of the following components:

- ▶ Base

The Base is the kernel of the toolkit. It includes the following elements:

- Module loader
- Language utilities (For example, array functions, functions for declaring classes and inheritance, and so forth)
- 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 an element

The standard Dojo profile (dojo.js) automatically loads the functionality above.

► Core

The Core provides additional facilities on top of the Dojo Base. This includes the following facilities:

- 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 (through Firebug lite)
- Build system
- Markup parser
- OpenAjax hub 1

► Dijit

Dijit is set of interaction-rich widgets and themes for use when developing Ajax applications. Dijit has the following features:

- Accessible (Keyboard support, support for high contrast mode, screen 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

2.4.2 Including the Dojo library

To extend your Web application with Dojo functionality, the first step is to download the latest build from the following Web page:

<http://dojotoolkit.org/downloads>

Extract the file in the WebContent directory and rename the root directory with dojo. Include the main dojo.js file in the dojo folder. For instance, this could be done in a index.jsp file, which is used to access the application with the browser.

```
<html>
  <head>
    <script type="text/javascript" src="dojo/dojo.js"></script>
  </head>
  <body>
  </body>
</html>
```

2.4.3 Dojo core library

This section describes some of the functionality provided by the Dojo core library that is used by the WebSphere Commerce Dojo Toolkit, and how it can be used to facilitate cross-browser JavaScript programming.

The Dojo 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. This scheme resembles 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”. 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 as follows:

- ▶ `dojo.provide(String moduleName)`

The `dojo.provide` function should be the first line of each JavaScript file you include in the application. It defines the module name of your JavaScript file and can be compared to Java’s `package` command. The function 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, suppose you have a directory called “myPackage” with a `myModule.js` inside. Further, suppose “myPackage” is located under the Dojo root directory (in this case, it is `WebContent`) at the same level as the `dojo` folder. 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 is added.

► `dojo.require(String moduleName)`

The `dojo.require` function is used to include the functionality of other JavaScript files and can be compared to Java's `import` command. This means you do not need to use script tags to include those files if 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 if 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 through a `dojo.require` statement, all dependent JavaScript files are also loaded. A typical JavaScript file using `dojo.require` is shown in Example 2-5.

Example 2-5 Typical JavaScript file using `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>
```

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. Do this in a separate script tag, or set the `djConfig` attribute in the script tag that includes the `dojo.js`. See Example 2-6.

Example 2-6 `djConfig` in the same script tag with `dojo.js`

```
<html>
  <script type="text/javascript" src="dojo/dojo.js"
    djConfig="parseOnLoad:true, isDebug: true"></script>
<body>
</body>
</html>
```

The following list details important djConfig parameters that can be set.

► **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 `baseUrl`. Dojo auto-populates `baseUrl`.

► **cacheBust**

If `cacheBust` is 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 you start the application a module could be loaded with `http://server:port/path/module.js?1210337274094`. The second time, it could be loaded 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 from the server and does not take it from the cache. This can be useful during module development.

► **debugAtAllCosts**

The default is `false`. If set to `true`, it can be used to debug JavaScript modules at the cost of performance. Because it slows down the client and has some unpredictable side-effects you should only use `debugAtAllCosts` 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 installed as a plug-in for the Firefox browser. `FirebugLite ()` provides similar debugging facilities as Firebug for other browsers like Internet Explorer®, Opera, and so forth. You can download `FirebugLite` at the following Web page:

<http://www.getfirebug.com/lite.html>

The Dojo Toolkit includes the `FirebugLite` library, which can be used with the `isDebug`, `debugContainerId`, `debugHeight`, `noFirebugLite`, and `pop-up` menu parameters.

In Firefox, if the Firebug plug-in is enabled, `console.*` function calls are 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, the following statements for other browsers apply.

In other browsers, when using Dojo, `console.*` functions can be called from all browsers without causing an error.

Depending on whether the following parameters are active, you may or may not be able to view the log messages.

- isDebug

The default setting is false. If set to true, Dojo adds a console window after the last HTML element on the page. Calls to console.* output to that window.

- debugContainerId

If set to an ID of an HTML element on the page (for example, a <div> with specific size and position on the page), the console window is inserted into that element instead of adding it after the last HTML element on the page.

- debugHeight

The height of the console window.

- popup

The default setting is false. If set to true, the console opens in a new window, 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 files to your domain. Set the dojoBlankHtmlUrl and the dojoIframeHistoryUrl to the path where you saved the files in your domain.

- ▶ locale, extraLocale

This parameter is needed for the internationalization (i18n) of displayed strings numbers, date, currencies, and so forth. If it is not set, the default value is navigator.userLanguage or navigator.language.

- ▶ modulePaths

This parameter is a map for setting module paths relative to djConfig.baseUrl. It can be used as an alternative to dojo.registerModulePath. 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.

- ▶ preventBackButtonFix

Set this parameter to false when using the Dojo back button module.

- **require**

Instead of using `dojo.require` in a script tag and loading modules separately, you can pass all module names in an array to `djConfig.require` and Dojo will load them automatically. For instance:

```
[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 the message, set `usePlainJson` to `true`. The default value is `false`. The `dojo.xhr` functions are discussed in 2.4.4, “XHR” on page 70.

In most situations, you only need to use `isDebug`, `parseOnLoad`, and `locale`, and allow Dojo to assign default values for the others.

Useful Dojo core functions

This section gives an overview of convenient functions available when programming with Dojo.

- **dojo.require**

This function is used to load a module and can be compared to Java’s `import` command. A module is always loaded once only, 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 returned. Example 2-7 shows the usage.

Example 2-7 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 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 2-8 shows the usage.

Example 2-8 Usage of `dojo.byId` and `dijit.byId`

```
<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 because some browsers, like Internet Explorer, do not support them. Example 2-9 on page 58 shows how to use each of the 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;
    });
</script>
```



```
    });  
    // res = [1, 2]  
</script>  
<body>  
</body>  
</html>
```

► **dojo.addOnLoad**

This function runs commands after the HTML body is loaded and Dojo initialization has completed. If you use the `onload` attribute in the `body` tag as in the following snippet, it is not guaranteed the Dojo initialization has completed:

```
<body onload="someFunc">
```

Use `dojo.addOnLoad` instead. Example 2-10 shows the usage of this function.

Example 2-10 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.toJson, dojo.fromJson**

The function `dojo.toJson` is used to serialize a JSON object into a string. The `dojo.fromJson` function is used to deserialize a string into a JSON object. These functions are shown in Example 2-11 on page 60.

```
<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 function is used to connect DOM events to function calls, or function calls to other function calls.
- ▶ **dojo.subscribe, dojo.publish**
This function is used to publish events and subscribe to events.
- ▶ **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.
- ▶ **dojo.isFunction, dojo.isArray, dojo.isObject**
These functions return true if the passed argument is of the type for which the function is checking. 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.
- ▶ **dojo.query**
This function invokes the powerful Dojo query engine.

The Dojo event system

Dojo comes with its own cross-browser event system, which 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.

Attaching event-handlers with dojo.connect

The `dojo.connect` function (Example 2-12) is provided by the Dojo base functionality. When you include the `dojo.js` file, you can start using the event system. You do not want to omit this in your Web application.

Example 2-12 Dojo connect function

```
dojo.connect(/*Object*/ obj, /*String*/ event, /*Object|null*/ context,  
/*String|Function*/ method, /*Boolean*/  
dontFix)
```

This function has the following parameters.

- ▶ **obj**: This is the source object that fires the event. If it is null, Dojo takes `dojo.global` as the source. `dojo.global` is a reference to the global scope in the browser. This parameter 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. Can be a reference to an object or `dojo.global`, if set to null.
- ▶ **method**: The method that is called when an 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.

This function has the following connecting events.:

Example 2-13 shows how to use the `dojo.connect` function to connect DOM events to function calls or connect one function call to another.

Example 2-13 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 object parameter, the events that can be fired and passed as the second parameter are the common JavaScript-events you would be connecting to without Dojo (For example, onclick, onblur, onfocus, onkeydown, onkeypress, and so forth).

► **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 (Example 2-14).

Example 2-14 dojo.disconnect

```
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 you pass to the source function are also passed to the handler function. See Example 2-15.

Example 2-15 Passing and accessing parameters

```
<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, the following cross-browser properties are added to the event object:

- 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 text box with event.preventDefault() as the first line, pressing a key will not cause the appropriate character to appear in the text box.

- 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. If used, it should be the first line in the event handler.

Dojo provides a map for keyboard key codes that improve the readability of the code. This is shown in Example 2-16.

Example 2-16 Keypress event

```

dojo.connect(dojo.byId("inp"), "onkeypress", function(e) {
    if(e.keyCode == dojo.keys.ENTER) {
        console.debug("Enter was pressed!");
    }
});

```

Note: All keys can be found at the following Web page:

<http://api.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.keys>

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 cover only the basics of DnD programming with Dojo.

Note: The full Dojo DnD documentation can be found at the following Web page:

http://docs.google.com/View?docid=d764479_11fcs7s397

To drag an element on the screen, mark its DOM node as a drag source. To drop an element (for instance, into a shopping cart), mark the area in which you want to drop as the drop target. Example 2-17 shows a basic example using two lists. Move names from the first list to the second by dragging a name into the second list.

Example 2-17 Drag and Drop

```
<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>
          here it can be any element you want to drag
        </div>
      </div>
    </div>
    <br/>
    <div dojoType="dojo.dnd.Target" accept="mySource">
      <div>drag target</div>
    </div>
  </body>
</html>
```

Add the attribute `dojoType` to the element and assign either `dojo.dnd.Source` or `dojo.dnd.Target` to it. For a drag source, 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 source you want to drop.

Handling the Back button problem

Using XMLHttpRequest for asynchronous client/server communication and dynamically updating parts of the browser screen has the drawback that the **Back** or **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 effort you can include the dojo.back module to add **Back** button functionality to your application. The idea is to set a state object with a back and forward callback function called when the user clicks either the **Back** or **Forward** button of the browser. Example 2-18 shows the state object.

Example 2-18 Back button state object 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, 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 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, then no fragment identifier is set.

To enable **Back** button functionality, set the preventBackButtonFix property of the djConfig variable to false. At application start, define an initial state object with dojo.back.setInitialState(state). Later, 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** switches between the states, calls the respective back or forward function, and changes the fragment identifier of the URL if needed.

Example 2-19 gives you an idea of how to implement **Back** button functionality in an application, and how to handle bookmarks used to access the application.

Example 2-19 Back button implementation

```
<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
        // existsvar 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. Use the **Back** and **Forward** buttons to undo or redo changes you made to the text boxes. Use bookmarks to jump to a specific state of the application. For instance, you can append the fragment identifier #Hi:there! and the first text box is populated with Hi and the second with there! When 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, insert the bookmark into the address bar and the application will jump to the previously bookmarked state. The application was successfully tested with Firefox 2.0+ and IE6.

Note: 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 the following Web page:

<http://dojotoolkit.org/book/dojo-book-0-9/part-3-programmatic-dijit-and-doj/back-button/browser-compatibility>

Bookmarks

The Madisons Store makes use of the `dojo.undo.browser` API for handling bookmarking. The idea is to add some type of unique identifier in the URL as actions are being performed on the page. Then, when loading the page, read that unique identifier in the bookmarked URL and figure out the state of the page.

Visit the following Web page for more details:

<http://manual.dojotoolkit.org/WikiHome/DojoDotBook/Book0>

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.

Example 2-20 is an example of the usage of the `dojo.query` function. A rich set of CSS3 selectors are supported.

Note: The list of supported selector types can be found at the following Web page:

<http://api-staging.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.query>

The list of supported f CSS3 selectors can be found at the following Web page:

<http://www.w3.org/TR/css3-selectors>

Example 2-20 Query 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.

Note: You can find a performance evaluation of the `dojo.query` function at the following Web page:

<http://dojotoolkit.org/node/336>

2.4.4 XHR

This section describes the Dojo functions that enable a client-server connection using XMLHttpRequest (XHR).

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 do not need to concern yourself with the different browser implementations of that object.

Dojo provides the following four 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)

This property is 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 forth). Furthermore, the `url` key can be used to read out static files on the server like normal text files.

- ▶ `content` (Object)

This property 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)

This property is the time to wait for the response in milliseconds. If the specified time passes, the `error/handle` callback is invoked.

- ▶ `form` (DOM node)

This property passes the node of a form on your Web page, or the ID of the form. Dojo 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 if a request is the same as a previous request, the browser does not send that request to the server. It 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. 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 as follows:

- `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`, 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)

This property is the callback function for a successful request. The callback is passed two arguments in the format function:

- 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

This property of `ioArgs` is the object that is passed to the `dojo.xhr` function.

- xhr

This property of `ioArgs` is the `xhr` object that is used for the request. It contains the properties `responseText`, `responseXML`, `status`, and `statusText`. If the response is valid XML, `responseText` is the response as a string and `responseXML` a reference to the parsed XML document. If the response is not XML, only the `responseText` is populated and `responseXML` is null. The property `status` is the HTTP status code, and `statusText` is the HTTP status text. A full list of HTTP statuses and texts can be found at the following Web page:

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

- url

This property of ioArgs is the URL that is used for the request.

- query

This property of ioArgs is the parameter that is passed in the request body. Only populated for POST and PUT.

- handleAs

This property of ioArgs is the data format used for parsing the server response.

► error (Function)

This property is the callback function for an unsuccessful request. The callback is passed two arguments in the format function:

- error

This argument is the 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 the error.dojoType is set to cancel or timeout.

- ioArgs

This argument is an object that contains information about the XHR call. It has the following properties:

- args

This property of ioArgs is the object that is passed to the dojo.xhr function.

- xhr

This property of ioArgs is the xhr object that is used for the request. It contains the properties responseText, responseXML, status, and statusText. If the response is valid XML, responseText is the response as a string and responseXML a reference to the parsed XML document. If the response is not XML, only the responseText is populated and responseXML is null. The property status is the HTTP status code, and statusText the HTTP status text. A full list of HTTP statuses and texts can be found at the following Web page:

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

- url

This property of ioArgs is the URL that is used for the request.

- query

This property of ioArgs is the parameter that is passed in the request body. Only populated for POST and PUT.

- handleAs

This property of ioArgs is the data format used for parsing the server response.

► handle (Function)

The handle callback is invoked for both a successful and unsuccessful request. Use this callback if you do not want to use either the load or error method. This callback is passed two arguments in the format function:

– 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

This ioArgs property is the object that is passed to the dojo.xhr function.

- xhr

This ioArgs property is the xhr object that is used for the request. It contains the properties responseText, responseXML, status, and statusText. If the response is valid XML, 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. The property status is the HTTP status code, and statusText the HTTP status text. A full list of HTTP statuses and texts can be found at the following Web page:

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

- url

This ioArgs property is the URL that is used for the request.

- query

This ioArgs property is the parameter passed in the request body. Only populated for POST and PUT.

- handleAs

This ioArgs property is the data format used for parsing the server response.

► synch (Boolean)

Set this property to true if you want the request to block. The default setting is false, which allows asynchronous communication with the server.

► headers (Object)

Use the headers property if you want to send additional HTTP header information. The format of the value is an object with key-value mappings.

Note: A list of all possible header definitions can be found at the following Web page:

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>

► contentType

This property sets the content type if you do not want to use the headers property for setting it.

xhrGet

Example 2-21 shows how to use the properties previously described with the xhrGet.

Example 2-21 xhrGet usage

```
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;
}
});
```

To have access to `xhrGet` inside the callback context, you have to use `dojo.hitch`. It is good practice always to 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, the error callback is also called and passed the JavaScript error.

Example 2-22 shows how to use the handle callback if you do not want to use load and error.

Example 2-22 Handle responses with 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 2-23 on page 77 shows how to transfer all of the form parameters to the server.

```
<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

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.

Note: For a detailed description of the `dojo.Deferred` object, check the API description at the following Web page:

<http://redesign.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.Deferred>

Example 2-24 gives an example 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 2-24 `dojo.Deferred` 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();
```

2.4.5 The Dojo Widget Library

Dijit is the Dojo 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.

Widget basics

A widget is constructed from JavaScript. Optional parts are an HTML template, CSS definitions, and images.

- ▶ JavaScript (*.js) file

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 forth)

Images can be used by the JavaScript file, in the HTML template, or in CSS style definitions. Imagine a widget as an HTML element that is already extended with functionality by Dojo (For example, a normal table enhanced by a sorting function or a text box that supports input validation).

Dojo widgets

This section provides an overview of the widgets that are available with the dijit module and which are used as a base for the WebSphere Commerce widgets described in 3.3, “WebSphere Commerce Framework for Dojo widgets” on page 134. The concepts of how to use widgets, both in a declarative and programmatic way are also presented.

Note: For a declarative or programmatic usage for all of the widgets, check the dijit/tests directory, the Dojo API at the following Web page:

<http://dojotoolkit.org/api>

To include a widget declaratively in your HTML markup, 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. See Example 2-25.

Example 2-25 Declaration of Dojo widgets

```
<html>
<head>
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<script type="text/javascript" src="../../dojo/dojo.js"
    djConfig="parseOnLoad: true, isDebug: true">
</script>
<script type="text/javascript">
    dojo.require("dojo.dnd.Source");
    dojo.require("dojo.parser");
</script>
</head>
<body class=tundra>
    <div dojoType="dojo.dnd.Source" class="source">
        SOURCE
    </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 you want to use. In Example 2-25 it is a Dojo DnD source. The class attribute is an argument that is passed to the `dnd.Source` widget when it is created, and represents the style used for this source. Depending on the widget type it may be necessary to pass additional arguments.

Tooltip widget

Example 2-26 shows a simple way of using Tooltip widget.

Example 2-26 Usage of a Tooltip widget

```
<html>
<head>
    <title>Dojo Tooltip Widget Test</title>
    <script type="text/javascript">
        var djConfig = {isDebug: true};
    </script>
    <script type="text/javascript" src="dojo/dojo.js"></script>
    <script type="text/javascript">
```

```

        dojo.require("dojo.widget.Tooltip");
    </script>
    <LINK href="theme/Master.css" rel="stylesheet" type="text/css">
</head>
<body>
<h1>Tooltip test</h1>
<p>Mouse-over the items below to test tooltips:</p>
<div style="overflow: auto; height: 100px; position: relative; border:
solid blue 3px;">
    <span id="s1">this is a html element for which we will display a
tooltip on mouse over</span><br>
</div>
<span dojoType="tooltip" connectId="s1" delay="100"
toggle="explode">tooltip information</span>
</body>
</html>

```

HorizontalSlider widget

Example 2-27 shows a simple way to use SliderHorizontal widget.

Example 2-27 Usage of a horizontal slider

```

<HTML>
<HEAD>
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<TITLE>Simple Slider Example</TITLE>
<script type="text/javascript" src="dojo/dojo.js"
djConfig="parseOnLoad: true"></script>
<script type="text/javascript">
    dojo.require("dojo.widget.*");
    dojo.require("dojo.event.*");
    dojo.require("dojo.widget.Slider");
</script>
</HEAD>
<BODY >
<b>Slider:</b>
<div id="horizontalSlider" dojoType="SliderHorizontal" initialValue="5"
    minimum="-10" maximum="50" snapValues="5"
    activeDrag="true"
    onValueChanged="dojo.byId('horizontalSliderValue').value
=arguments[0];"
    buttonStyle="top:1px;"
    backgroundStyle="padding:8px 4px 8px 4px;border:1px solid red;"
    backgroundSize="width:500px;height:5px;"

```

```

        backgroundSrc="src/widget/templates/images/blank.gif"
        progressBackgroundSrc="src/widget/templates/images/slider-bg.gif"
        handleStyle="top:0px;width:13px;height:18px;"
        handleSrc="src/widget/templates/images/slider.gif"
        "style="border:solid blue;
        border-width:16px 12px 8px 4px;padding:4px 8px 12px
        16px;margin:9px;">
    </div>
<br/>
Horizontal Slider Value:
<input id="horizontalSliderValue" name="horizontalSliderValue"
value="0"/>
<button onclick="dojo.widget.byId ('horizontalSlider').setValue
    (dojo.byId('horizontalSliderValue').value)">set
</button>
<i>(updates when slider handle is moved)</i>
<br />
<input
    id="horizontalSliderValue" name="horizontalSliderValue" value="0"/>
    <button onclick="dojo.widget.byId ('horizontalSlider').setValue
    (dojo.byId('horizontalSliderValue').value)">set</button> <i>(updates
    when slider handle is moved)</i>
<br/>
</BODY>
</HTML>

```

The <div> tag creates a horizontal slider, setting all its properties. Although moving the slider from left to right, the value of the horizontalSliderValue element changes with the current value. The slider values can be changed by clicking the **SET** button, which will take the value from the horizontalSliderValue element.

Button widget

Dijit has a few button widgets like a simple click button dijit.form.Button, a DropDownButton, and much more. Example 2-28 shows how can we use dijit.form.Button widget.

Example 2-28 Usage of a dijit.form.Button

```

<html>
<head>
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<script type="text/javascript" src="../../dojo/dojo.js"
    djConfig="parseOnLoad: true, isDebug: true">
</script>

```



```
<script type="text/javascript">
    dojo.require("dijit.form.Button");
    dojo.require("dojo.parser");

    function myClick()
    {
        alert('My First Button');
    }
</script>
</head>
<body>
    <div>
        <button dojoType="dijit.form.Button"
onClick="myClick();">click</button>
    </div>
</body>
</html>
```

Button widget attributes

The following list details the button widget attributes:

- ▶ **label**
You can set text on a button.
- ▶ **iconClass**
You can display an image on the button and set up a CSS class which uses that icon as a background-image.

Button widget methods

The button widget method is `onClick`. The function is called when a button is clicked. The user can override this function for adding extra functionality before submitting a form.

DropDownButton widget

Dijit offers the `DropDownButton` widget to provide a drop-down menu when clicked.

Example 2-29 on page 84 shows a way to use `dijit.form.DropDownButton`, `dijit.Menu` and `dijit.MenuItem` to create the drop-down menu in Figure 2-19 on page 85.

```
<html>
<head>
<style type="text/css">
    @import
    "http://o.aolcdn.com/dojo/1.0.0/dijit/themes/tundra/tundra.css";
    @import
    "http://o.aolcdn.com/dojo/1.0.0/dojo/resources/dojo.css"
</style>

<script type="text/javascript"
src="http://o.aolcdn.com/dojo/1.0.0/dojo/dojo.xd.js"
    djConfig="parseOnLoad: true">
</script>
<script type="text/javascript">
    dojo.require("dojo.parser");
    dojo.require("dijit.form.Button");
    dojo.require("dijit.Menu");
    function copyFunction() {
        alert("Copy action starts!!");
    }
    function pasteFunction() {
        alert("Paste action starts!!");
    }
</script>
</head>
<body class="tundra">
    <div dojoType="dijit.form.DropDownButton">
        <span>Edit</span>
        <div dojoType="dijit.Menu" id="Edit">
            <div dojoType="dijit.MenuItem" label="Copy"
                onclick="copyFunction();"></div>
            <div dojoType="dijit.MenuItem" label="Paste"
                onclick="pasteFunction();"></div>
        </div>
    </div>
</body>
</html>
```

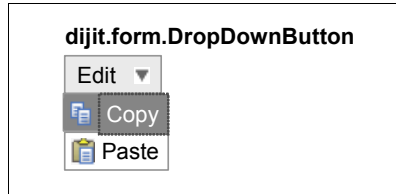


Figure 2-19 DropDownButton widget in use

DropDownButton, for example, may be used in Web applications for creating a menu for a rich text editor.

DropDownButton widget attributes

The following list details the DropDownButton widget attributes:

- ▶ **iconClass**
This attribute describes the class to apply to a <div> in a button to make it display a style.
- ▶ **label**
This attribute describes the text to display in button.
- ▶ **optionsTitle**
This attribute describes the text that describes the options menu.
- ▶ **showLabel**
If the showLabel attribute is set to true, it will display the text label in a button.

DropDownButton widget methods

The following list details the DropDownButton widget methods:

- ▶ **addChild**
This method inserts a given DOM node as a child of the current DOM node.
- ▶ **getChildren**
This method returns an array of children widgets.
- ▶ **setLabel**
This method sets the label of the button with the given value.
- ▶ **hasChildren**
This method returns true if widget has children.
- ▶ **removeChild**
This method removes the given widget instance from the current widget but does not destroy it.

► onClick

This method is called when button is clicked. A user can override this function for adding some extra functionality before submitting a form.

Dialog and TooltipDialog widgets

Example 2-30 show a usage of the dijit.Dialog widget to display forms overlaying the current page.

Example 2-30 Dialog widget

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>Dialog demo</title>
<style type="text/css">
@import "http://o.aolcdn.com/dojo/1.0/dijit/themes/tundra/tundra.css";
@import "http://o.aolcdn.com/dojo/1.0/dojo/resources/dojo.css"
</style>
<script type="text/javascript"
src="http://o.aolcdn.com/dojo/1.0/dojo/dojo.xd.js"
djConfig="parseOnLoad: true">
</script>
<script type="text/javascript">
    dojo.require("dojo.parser");
    dojo.require("dijit.form.Button");
    dojo.require("dijit.Dialog");
    dojo.require("dijit.form.TextBox");
</script>
</head>
<body class="tundra">
<button dojoType="dijit.form.Button"
onclick="dijit.byId('dialog1').show()">show dialog</button>
<div dojoType="dijit.Dialog" id="dialog1" title="First Dialog" >
<table>
    <tr>
        <td><label for="name">First Name: </label></td>
        <td><input dojoType="dijit.form.TextBox" type="text"
name="firstName"></td>
    </tr>
    <tr>
        <td><label for="loc">Last Name: </label></td>
        <td><input dojoType="dijit.form.TextBox" type="text"
name="lastName"></td>
    </tr>
</tr>
</table>
```

```

        <tr>
            <td colspan="2">
                <button dojoType=dijit.form.Button type="submit">OK</button></td>
            </tr>
        </table>
    </div>
</body>
</html>

```

Example 2-31 show a simple usage of dijit.TooltipDialog widget to display forms in a tooltip.

Example 2-31 TooltipDialog widget

```

<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>TooltipDialog demo</title>
<style type="text/css">
@import "http://o.aolcdn.com/dojo/1.0/dijit/themes/tundra/tundra.css";
@import "http://o.aolcdn.com/dojo/1.0/dojo/resources/dojo.css"
</style>
<script type="text/javascript"
src="http://o.aolcdn.com/dojo/1.0/dojo/dojo.xd.js"
djConfig="parseOnLoad: true"></script>
<script type="text/javascript">
    dojo.require("dojo.parser");
    dojo.require("dijit.form.Button");
    dojo.require("dijit.Dialog");
    dojo.require("dijit.form.TextBox");
</script>
</head>
<body class="tundra">
<div dojoType="dijit.form.DropDownButton">
<div dojoType="dijit.TooltipDialog" id="dialog1" title="Second Dialog">
<table>
    <tr>
        <td><label for="name">First Name: </label></td>
        <td><input dojoType="dijit.form.TextBox" type="text"
name="firstName"></td>
    </tr>
    <tr>
        <td><label for="loc">Last Name: </label></td>
        <td><input dojoType="dijit.form.TextBox" type="text"
name="lastName"></td>
    </tr>
</table>
</div>
</div>

```

```

        </tr>
        <tr>
            <td colspan="2" align="center">
                <button dojoType=dijit.form.Button
type="submit">OK</button></td>
            </tr>
        </table>
    </div>
</div>
</body>
</html>

```

DnD widget

Example 2-32 shows a way to use a DnD widget. We have two images that can be dragged onto a target.

Example 2-32 Dnd Dojo widget

```

<html>
<head>
<title>Simple DnD Example</title>
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<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>
</head>
<body style="font-size: 12px;">
<h1>A Simple Example</h1>
<table><tbody><tr>
<td>
<!-- Create a source with two nodes -->
<div dojoType="dojo.dnd.Source" jsId="c1" class="source">
    SOURCE
    <div class="dojoDndItem" dndType="first">
        
    </div>
    <div class="dojoDndItem" dndType="second">
        
    </div>
</div>
</td>

```

```

<td>
<!-- Create a target that accepts nodes of type first and second. -->
<div dojoType="dojo.dnd.Target" jsId="c2" class="target"
accept="first,second">
    TARGET
</div>
</td>
</tr><tbody/></table>

```

The outermost <div> tag creates a source object. The inner <div> tags with class dojoDndItem create nodes that can participate in DnD. The dndType attribute can be used to specify the type of a node. The accept attribute of the Target is a comma-separated list of types that can be dragged onto this node. This means that only those nodes whose dndType is present in the list can be dragged onto this target. You can restrict DnD operations by specifying appropriate types for nodes and targets.

Menu, MenuItem, and PopupMenuItem widgets

The Menu widget can be used for creating a context menu, or with DropDownButton widgets. MenuItem widgets are the actual items in the menu. The PopupMenuItem also creates an item in a menu, but displays a submenu or other widget to the right. Figure 2-20 and Example 2-33 on page 90 shows an example of how you can use all these widgets to create a menu.

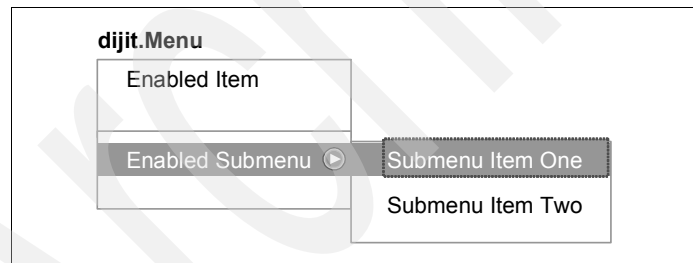


Figure 2-20 Menu, Menu Item, and PopupMenuItem widgets in use

```
<html>
<head>
<style type="text/css">
    @import
    "http://o.aolcdn.com/dojo/1.0.0/dijit/themes/tundra/tundra.css";
</style>
<LINK href="theme/Master.css" rel="stylesheet" type="text/css">
<script type="text/javascript" src="dojo/dojo.js"
    djConfig="parseOnLoad: true">
</script>
<script type="text/javascript">
    dojo.require("dojo.parser");
    dojo.require("dijit.Menu");
</script>
</head>
<body class="tundra">
<div dojoType="dijit.Menu" id="EnabledMenu"
    contextMenuForWindow="true" style="display: none;">
    <div dojoType="dijit.MenuItem" iconClass="myStyle"
        onClick="alert('Item is enabled!!!');">Enabled
Item</div>
    <div dojoType="dijit.PopupMenuItem" id="Enabled submenu">
        <span>Enabled submenu</span>
        <div dojoType="dijit.Menu">
            <div dojoType="dijit.MenuItem"
                onClick="alert('Submenu Item One!!!!')">Submenu Item
One</div>
            <div dojoType="dijit.MenuItem"
                onClick="alert('Submenu Item Two !!!')">Submenu Item
Two</div>
        </div>
    </div>
</body>
</html>
```

dijit.Menu attributes

The following list details the dijit.Menu attributes:

- ▶ contextMenuForWindow
If this attribute is set to true, you can right-click anywhere on the window to open the context menu. If this attribute is set to false, specify targetNodeIds
- ▶ popupDelay -
This attribute sets the number, in milliseconds, it takes to hover over a link for it to open automatically.
- ▶ targetNodeIds
This attribute gives to an array of DOM nodes an ID. This should be filled with nodeIds upon widget creation, and will become a context menu for those nodes.

dijit.Menu methods

The following list details the dijit.Menu methods:

- ▶ addChild
This method inserts the given DOM node as a child of the current DOM node.
- ▶ bindDomNode
This method attaches a menu to the given node.
- ▶ getChildren
This method returns an array of children widgets.
- ▶ removeChild
This method removes the given widget from the current widget but does not destroy it.
- ▶ unBindDomNode
This method detaches a menu from the given node.
- ▶ onClose
This method is called when the current menu is closed.
- ▶ onOpen
This method is called when the current menu is open by mouse events.

dijit.Menuitem attributes

The following list details the dijit.Menuitem attributes:

- ▶ disabled
If this attribute is set to true, the menu item is disabled. If this attribute is set to false (the default value), the menu item is enabled.
- ▶ iconClass
This attribute is a style class that can be applied to a <div> in a button to make it display an icon.

dijit.Menuitem methods

The following list details the dijit.Menuitem methods:

- ▶ setDisabled
This method enables or disables this menu item.
- ▶ getParent()
This method returns the parent widget of this widget.
- ▶ onClick()
This method is called to handle click events.

.dijit.MenuSeparator

The .dijit.MenuSeparator creates a line between two menu items.

Note: For more details about Dojo toolkit, refer the following Web page:

<http://dojotoolkit.org>

The Web 2.0 Store solution uses the new Dojo 1.0 toolkit. There are many enhancements and added features available in Dojo 1.0, including better performance and full accessibility support. The widgets that were supplied in the previous Web 2.0 Store has been recalibrated to use the Dojo 1.0 code base. The first attempt is to use the Web 2.0 feature pack that is a part of WebSphere Application Server. Otherwise, use the toolkit that is a part of Commerce.

Note: Web 2.0 Store Dojo toolkit is described in 3.3, “WebSphere Commerce Framework for Dojo widgets” on page 134.

Design and implement your Web 2.0 Store

This chapter explains how to implement your own Web 2.0 Store using Ajax and Dojo and the Madisons Store (Web 2.0 reference application for WebSphere Commerce). Developers can take advantage of this and the next chapters in combination of our reference application to build their own Web 2.0 Store in a more efficient way, this helps developers quickly design, develop, test, analyze, and deploy high-quality Web 2.0 Stores.

This chapter has the following sections:

- ▶ Section 3.1, “WebSphere Commerce Web 2.0 Store structure” on page 94 explains the package, file content structure, and page layout of the Madisons Store. This information assists the developer in getting familiar with the store structure, and understanding how the file contents and page layouts are organized.
- ▶ Section 3.2, “WebSphere Commerce Ajax framework for Dojo” on page 105 details the WebSphere Commerce Ajax framework, which is an extension of Dojo Ajax and event API. This section, of particular interest to developers, shows how easy it is to compose a Web 2.0 Store using the WebSphere Commerce framework. This section has examples and sample code available to the reader.

- ▶ Section 3.3, “WebSphere Commerce Framework for Dojo widgets” on page 134 deals with Dojo functionality and the Dojo widgets used in our store. This section is key for those who want to implement, extend, or customize Dojo widgets to fit their application. This section explains each widget used in our application and gives examples how to extend or customize the widget to fit their requirements.
- ▶ Section 3.4, “Introduce Web 2.0 features into your Web 1.0 Store” on page 183 explains how the Web 2.0 features can be added to a Web 1.0 Store. This section is aimed at users who want to accelerate their Web 1.0 Store by adding some Web 2.0 features, without developing and publishing a new store archive.

3.1 WebSphere Commerce Web 2.0 Store structure

This section explains the basic package, file content, and page layout structure you should use as part of a best practice store structure.

3.1.1 WebSphere Commerce package structure for Web 2.0 Store

The Web 2.0 Store has a typical Webapp package structure. We have two different packages:

- ▶ for store JavaServer™ pages (JSPs) and snippets
- ▶ for Dojo

These two folders comes under the following directory:

```
*Runtime
WC_installdir/AppServer/profiles/demo/installedApps/WC_demo_cell/WC_
demo.ear/Stores.war/
* WebSphere Commerce Developer
WCDE_installdir/workspace/Stores/WebContent/stores
```

The typical Web 2.0 Store directory contains the following directories:

- ▶ css

This directory contains the style sheets for the store. You can have style sheets for different browsers if needed, and can store language-specific style sheets.

- ▶ images

This directory contains the images used in the UI. If Dojo widgets have any images for the UI, you can put them in this folder and have a reference in the style sheet.

- ▶ include
This directory contains all common layout snippets for headers, footers, and sidebars.
- ▶ javascript
This directory contains most of the store java script in the .js files.
- ▶ shoppingarea
This directory contains the top level JSPs.
- ▶ snippets
This directory contains the code snippets commonly used at multiple places and multiple pages.
- ▶ userarea
This directory contains pages related to userarea like MyAccount, Wishlist, and so forth.

The Dojo package contains dijit, Dojo, Dojox, and WebSphere Commerce sub-packages. Find more information about these sub-packages in 2.4, “Web 2.0 application concepts” on page 49, WebSphere Commerce contains WebSphere Commerce Ajax APIs and extended Dojo widgets.

3.1.2 File content structure

Basic top-level JSP structure will contain basic layout structure to maintain header, footer, sidebar, and main content, and the inclusions and imports of other parts of the code. Example 3-6 on page 99 show the basic structure of the JSP. We maintain code level modularity to allow for more flexible, reusable, and easily maintainable code. Well-structured and modularized code is useful with large-scale of code snippets. It impacts the total cost of implementation (TCOI) as well, because it is easy to maintain. As a Web 2.0 Store best practice, we suggest you maintain the same code and file structure.

Using external JavaScript

We maintain the JavaScript code away from the JSP file contents, and keep them in a separate .js files. We refer to the code with a single line statement in the JSP where that JavaScript code is needed. In this way, the code is reused wherever it is needed, and the code can be easily maintained. Putting the JavaScript code with the JSP code makes it tough for a new developer to understand the flow, so external JavaScript is a convenient and easily-readable formatting choice for new developers.

External JavaScript ease of use

Example 3-1 shows how to include a JavaScript file in a JSP.

Example 3-1 Including the JavaScript file in the JSP

```
<script type="text/javascript" src="<c:out  
value="{jspStoreImgDir}javascript/ErrorHandler.js"/>"></script>
```

Example 3-1 is a one line statement that provides the ErrorHandler JavaScript code. In a page needing the ErrorHandler, use this statement to render the code. JavaScript external files provide the advantage of outsourcing JavaScript code to external files.

External JavaScript performance

By putting the JavaScript into a separate external .js file, you separate the JavaScript from the actual content of the page, which improves the page load time. Instead of putting the JavaScript content in the JSP, you can put that content into the JavaScript file and render. This helps improve site performance. We have all our JavaScript files under the following directory.

```
*Runtime  
WC_installdir/AppServer/profiles/demo/installedApps/WC_demo_cell/WC_  
demo.ear/Stores.war/javascript/  
* WebSphere Commerce Developer  
WCDE_installdir/workspace/Stores/WebContent/stores/javascript/
```

Under this folder, we have some .js files that are page-specific, and some .js files that are used in multiple pages (like Common.js, ErrorHandler.js, ServicesDeclaration.js, CommonContextsDeclarations.js, CommonControllersDeclaration.js).

If your page contains less code, search engines can index your page much faster and much better. Search engines do not actually look for the UI part, or the logical part of the page, they look for titles, alt tags, names, and so forth. Search engines are not concerned about JavaScript, so moving code makes it easier for them and easier for site visitors. The less content in your page, the faster it loads and indexes the site.

Do not use this same inclusion of the JavaScript file in the top level and sub-level JSPs. You need to include only once in the top level JSP, so it is lightweight, with a reduced page load time. To avoid the multiple inclusions of the JavaScript file, we have a special statement, as shown in Example 3-2 on page 97, in the JavaScript file itself.

```
if(typeof(ErrorHandler) == "undefined" || !ErrorHandler ||
!ErrorHandler.topicNamespace){
ErrorHandler = {
//All the JS content goes here
}
}
```

In Example 3-2, we have a condition that checks for the existence of the ErrorHandler object. If it already exists, it will not render the same content again. This is useful when maintaining the same object instance for the entire page flow.

Using External CSS

If you have cosmetic style properties as part of your code, you must change your code every time you change styles. It is difficult for a new developer to find the exact statement where he needs a style change. Instead, separate the styles from the actual code and maintain them in separate CSS files. This way, you need update only one file to update all of your Web site's pages. This may change when you have browser-specific CSS content.

External CSS ease of use

CSS files are easy to understand and read, so in-depth technical skills are not required to change styles. We used relevant class attributes for the styles. This helps to identify the right class for the right element.

External CSS makes maintaining your site easy. Changing the font of an item name is a matter of changing a property in one class in the CSS. This one change affects every instance of that item name. This makes for easy implementation and cost-effective maintenance without changing your actual code, so the logic still works correctly.

You do not need any special editors to view and change an external CSS. It can be opened and edited using any simple text editors. You can change your store's look by changing one CSS. Do not worry if you do not like the new look, just undo the changes to revert to the old style.

External CSS performance

There are definite performance advantages to using an external CSS. Because the search engine does not look at your UIs and the widgets in your code, it can more quickly scan the code for the titles and names it requires to index your page. Keeping an external CSS, therefore, is beneficial to you from a customization, maintenance and performance perspectives.

Example 3-3 on page 98 shows how to include the external CSS in your code.

Example 3-3 Including external CSS in the JSP

```
<link rel="stylesheet" href="<css file here>" type="text/css"/>
```

Use this statement wherever you use the same style. Set the relative file path and CSS file name as in Example 3-4.

Example 3-4 Setting the relative file path and CSS name in an external CSS

```
<link rel="stylesheet" href="<c:out  
value="{jspStoreImgDir}{vfileStylesheet}"/>" type="text/css"/>.
```

Sometimes you may need browser-specific CSS properties. In this case, we need to have browser-specific classes. This is done with multiple external CSS files. While including the files, we can have the browser to check the condition for specific classes. If the condition is true, it overwrites the class. Example 3-5 shows the conditional inclusion of multiple CSS files.

Example 3-5 Conditional inclusion of browser-specific CSS

```
<link rel="stylesheet" href="<c:out  
value="{jspStoreImgDir}{vfileStylesheet}"/>" type="text/css"/>  
<!--[if lte IE 6]>  
<link rel="stylesheet" href="<c:out  
value="{jspStoreImgDir}{vfileStylesheetie}"/>" type="text/css"/>  
<![endif]-->
```

There are two CSS files in Example 3-5. The first one is unconditional and is the same for all browsers. The second one is included only when the browser is Internet Explorer version 6 or earlier. This allows for multiple inclusions and browser-specific inclusions in your pages.

Modular code

We have complex functional flows in our store and we have a large amount of code for these all functional flows. It is difficult to organize the code as a single piece. The whole site cannot be developed by a single developer. There are multiple developers involved in the process. During development they may need some functionality common across every page. Because there is no common code sharing between pages, however, there is a redundancy penalty. When the product goes live and the pages need some common change, every page must be changed. This takes time and resources. If the developer is new to the pages, unplanned errors may result.

To avoid confusion, and to achieve easy-to-use, flexible, reusable code, we modularized our code. You will not find much complex code in any of the top-level JSP. It contains the inclusions and references to other code snippets. Each of these snippets are self-organized with some input data. Just include or import them with the initial data they require. This concept is easier to grasp once you start coding this way. We have a large amount of complex code for our store. We organized it to be as modular as possible. The following examples (Example 3-6 to Example 3-12 on page 101) detail our main top-level JSP code.

Note: These examples are in the same sequence as the actual file. Find out how to download the full content of the `TopCategoriesDisplay.jsp` file in Appendix A, “Additional material” on page 237.

Example 3-6 Main comment part of code

```
<%--
    *****
    * This JSP page displays the store's top-level categories. It is used
    as the starter store's homepage.
    * It imports three JSP pages:
    * - CachedHeaderDisplay.jsp, which displays the header of the page
    * - CachedTopCategoriesDisplay.jsp, which displays the top-level
    categories
    * - CachedFooter.jsp, which displays the footer of the page
    *****
--%>
```

About Example 3-6: This part of the code explains what the JSP shows, what it contains, and other references to the code snippets. If the file is a code snippet or sub-JSP, you can list the parameters and inputs needed.

Example 3-7 Main inclusions in the page

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://commerce.ibm.com/base" prefix="wcbase" %>
<%@ taglib uri="http://commerce.ibm.com/foundation" prefix="wcf" %>
<%@ taglib uri="flow.tld" prefix="flow" %>
<%@ include file="../../include/JSTLEnvironmentSetup.jspf"%>
<%@ include file="../../include/nocache.jspf"%>
```

About Example 3-7: This part of the code contains the basic inclusions for functionality, taglibraris, and environment.

Example 3-8 Stylesheet references

```
<link rel="stylesheet" href="<c:out
value="\${jspStoreImgDir}\${vfileStylesheet}"/>" type="text/css"/>
<!--[if lte IE 6]>
<link rel="stylesheet" href="<c:out
value="\${jspStoreImgDir}\${vfileStylesheetie}"/>" type="text/css"/>
<![endif]-->
```

Example 3-9 Dojo file inclusion and other JavaScript file inclusions

```
<script type="text/javascript" src="<c:out value="\${dojoFile}"/>"
djConfig="\${dojoConfigParams}"></script>
<script type="text/javascript" src="<c:out
value="\${jspStoreImgDir}javascript/Common.js"/>"></script>
<script type="text/javascript" src="<c:out
value="\${jspStoreImgDir}javascript/CategoryDisplay.js"/>"></script>
<script type="text/javascript" src="<c:out
value="\${jspStoreImgDir}javascript/ErrorHandle.js"/>"></script>
<script type="text/javascript">
    dojo.require("wc.widget.RefreshArea");
    dojo.require("wc.render.RefreshController");
    dojo.require("wc.render.Context");
    dojo.require("dojo.parser");
    dojo.require("dojo.dnd.Source");
    dojo.require("wc.widget.ScrollablePane");
    dojo.require("dijit.layout.ContentPane");
</script>
```

Example 3-10 Setting the messages

```
<script type="text/javascript">
categoryDisplayJS.setCommonParameters('${WCPParam.langId}', '${WCPParam.st
oreId}', '${WCPParam.catalogId}');
    ErrorHandler.setErrorMessage("ERR_RESOLVING_SKU", "<fmt:message
key='ERR_RESOLVING_SKU' bundle='${storeText}'/>");
    ErrorHandler.setErrorMessage("QUANTITY_INPUT_ERROR", "<fmt:message
key='QUANTITY_INPUT_ERROR' bundle='${storeText}'/>");
    ErrorHandler.setErrorMessage("WISHLIST_ADDED", "<fmt:message
key='WISHLIST_ADDED' bundle='${storeText}'/>");
    ErrorHandler.setErrorMessage("SHOPCART_ADDED", "<fmt:message
key='SHOPCART_ADDED' bundle='${storeText}'/>");
</script>
```

About Example 3-10: You can set error messages, alert messages, and successful messages in a JavaScript array that is used during Ajax calls.

Example 3-11 service, RefreshController, and RenderContext declarations

```
<script type="text/javascript" src="<c:out  
value="\${jspStoreImgDir}javascript/ServicesDeclaration.js"/>"></script>  
<script type="text/javascript">
```

```
ServicesDeclarationJS.setCommonParameters('${WParam.langId}', '${WPara  
m.storeId}', '${WParam.catalogId}');  
</script>
```

About Example 3-11: This section contains code for the service declarations, RefreshController and render context declarations, and sets all require parameters to those objects. As part of the best practice we have these declarations in form of JavaScript to improve the performance of the page and to give minimum loading time.

Example 3-12 Main page content

```
<div id="page">  
  <!-- Header Nav Start -->  
  <%@ include file="../../include/LayoutContainerTop.jspf"%>  
  <%@ include file="../../include/BreadCrumbTrailDisplay.jspf"%>  
  <!-- Header Nav End -->  
  
  <!-- Main Content Start -->  
  <div id="content_wrapper">  
  
    <div id="left_nav">  
      <%@ include file="../../include/LeftSidebarDisplay.jspf"%>  
    </div>  
  
    <div id="right_nav">  
      <%@ include  
file="../../include/HomeRightSidebarDisplay.jspf"%>  
    </div>  
  
    <!-- Content Start -->  
    <div id="MessageArea" >  
      <BR>  
      <span id="ErrorMessageText" class="red" tabindex="-1">
```

```

        </span>
        <BR><BR>
    </div>
    <%out.flush();%>
    <c:import
url="${jspStoreDir}Snippets/Catalog/CategoryDisplay/CachedTopCategories
Display.jsp">
        <c:param name="storeId" value="${WCPParam.storeId}"/>
        <c:param name="catalogId" value="${WCPParam.catalogId}"/>
        <c:param name="langId" value="${langId}"/>
        <c:param name="showLanguageCurrency" value="true"/>
        <c:param name="showContractDisplayCustomization"
value="false"/>
    </c:import>
    <%out.flush();%>

    <!-- Content End -->
</div>
<!-- Main Content End -->
<!-- Footer Start Start -->
<%@ include file="../../include/LayoutContainerBottom.jspf"%>
<!-- Footer Start End -->
</div>

```

About Example 3-12: This code represents the main content for the page. This section maintains different layouts (like headers, footers, sidebars, and the main center content of the page).

3.1.3 Page layout structure

Our basic page layout (Figure 3-1 on page 103) has the following five sections:

- ▶ Header
- ▶ Footer
- ▶ Left sidebar
- ▶ Right sidebar
- ▶ Main content

For any Ajax action on the page, we either refresh a section or part of a section. This approach provides faster interaction between client and the server, and improves server performance.



Figure 3-1 Basic page layout

Each of these five sections are taken care by separate code modules. On the top-level JSP, we include each of these modules separately, as shown in Example 3-6 on page 99 to Example 3-12 on page 101. Each of these sections contains subsections. Headers, footers, and sidebars are common content across pages, so we made these sections as reusable snippets across pages. Header and footer information is consistent through the pages, but the sidebar content can change from page to page. The Header shows the Title of the store, search box, mini shopping cart, top category drop-down menus, and other common links (like Home, login, my account and so forth).

The BreadCrumbTrailDisplay widget, which tracks the path followed to reach the current page, is also part of header. The footer contains the common links to the other store pages (such as orderstatus, MyAccount, and Help).

The left sidebar content varies across pages. For catalog browsing it shows the top and subcategories. For the MyAccount page it shows sublinks to the MyAccount page (such as personal information, wishlist, orderstatus, and address book). For Fast Finder, it shows the brand selections and priceranger. The right sidebar displays recommendations, and different e-marketing spots based on the category or product selection. Some store pages (such as MyAccount, checkout page, and so forth) may not contain all sections. See Figure 3-2 for an example. The MyAccount page does not contain a right sidebar, because the right sidebar is meant for recommendations based on the catalog selection. An account page would not have recommendations.

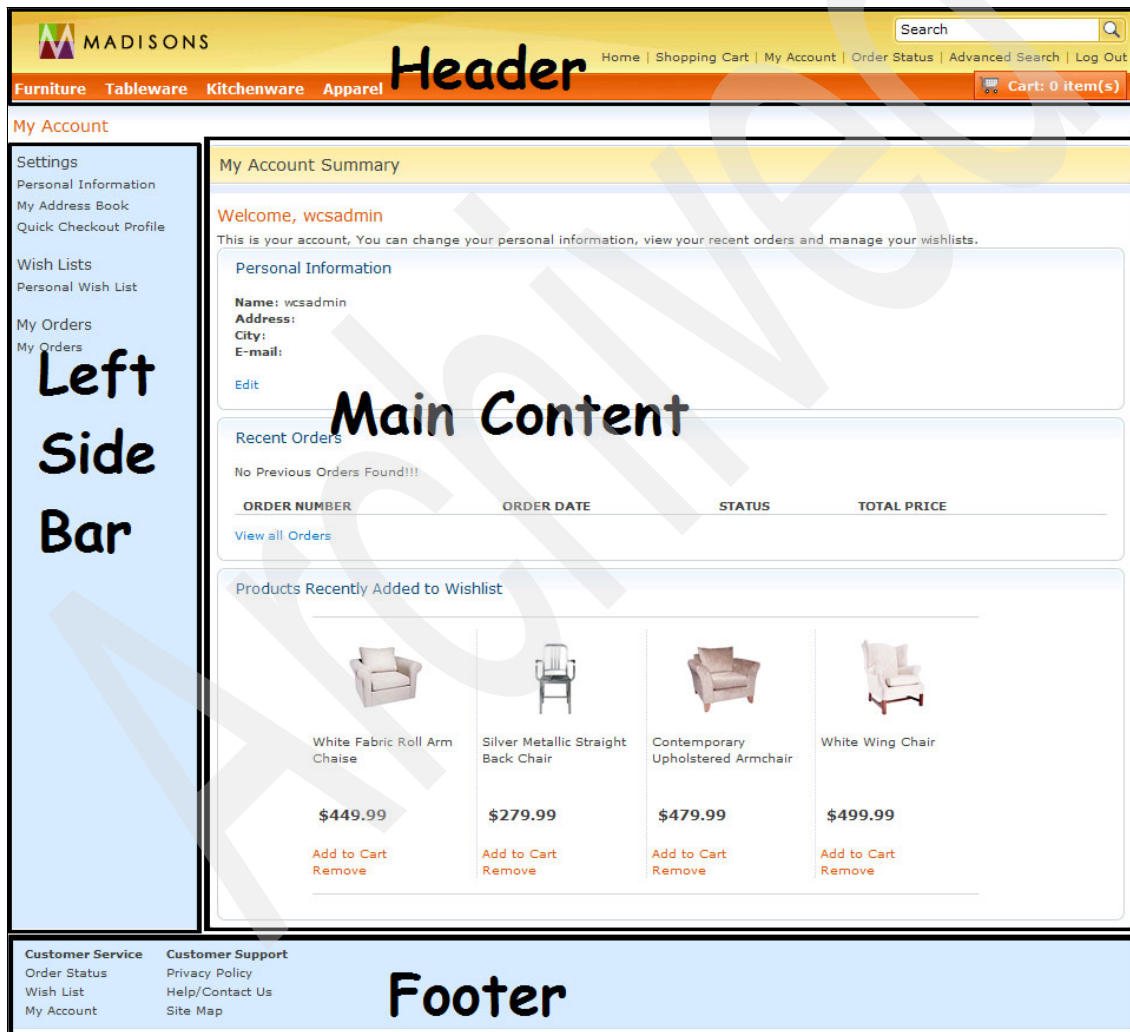


Figure 3-2 MyAccount page layout without right sidebar

3.2 WebSphere Commerce Ajax framework for Dojo

The WebSphere Commerce Ajax framework is an extension of the Dojo Ajax and events API, which provides an easy to use framework that meets most Ajax requirements for storefront development. It hides some of the complexity and repetitive code, and provides an easy-to-use and fitting functionality for the storefront.

We have two types of update requests to the server:

- ▶ For WebSphere Commerce Controller

These commands are well-known, as we use it in our all starter stores.

- ▶ For WebSphere Commerce Component Services

There are four basic scenarios that include the requirements followed by WebSphere Commerce Ajax framework functionality, which addresses these requirements and explores the best places for each of these functionalities. The following four scenarios cover most of our store requirements.

- ▶ Make an Ajax call to WebSphere Commerce server to refresh a section in the page because of certain shopper interactions.
- ▶ Make an Ajax call to the WebSphere Commerce server to update some business object. If the update is successful, a number of different areas in the Web page need to be refreshed with new content. The new contents for the page are retrieved through subsequent Ajax calls to the WebSphere Commerce server.
- ▶ Make an Ajax call to the WebSphere Commerce server for JSON data and then update certain areas in the page using a JavaScript DOM manipulation API.
- ▶ A combination of the second and third scenarios. In this scenario instead of wanting to make the code modular, less server traffic is desired. Make an Ajax call to WebSphere Commerce server to update some business object, and if the update is successful, the relevant information to refresh the contents of the page are returned as a JSON object. The client side then has the contents of the JSON object and uses DOM manipulation API to modify the areas in the Web page that change as a result of the update.

Note: The third and fourth scenarios do not require the use of the WebSphere Commerce Ajax framework because everything can be done using the Dojo Ajax API.

The next four subsections explain the above scenarios, providing in-depth information about the scenarios and the API required to use for each scenario.

3.2.1 Getting familiar with the Ajax Framework extension

This section explains the Ajax Framework extension and the JavaScript objects we use in these scenarios. We use refresh controller, Render Context, service JavaScript objects, and the wcf:json tag from the framework. And in the next section, you will see how we use these objects to fulfill each scenario.

Using this framework, you can refresh or reload a part of the page based on a business object update, or a simple loading of new content in place of old content. The part of the page to update is called RefreshArea. This RefreshArea is a simple <div> element registered to a RefreshController. RefreshController is the central point of control for the RefreshAreas registered to this controller. The Render Context object keeps track of client context information and can trigger renderContextChanged events whenever updates occur to a property in the Render Context object. The refresh controllers are automatically registered to listen to the renderContextChanged events. The RefreshController logic determines if the context change should trigger an update of a RefreshArea widget. Therefore, in the refresh controller's renderContextChangedHandler, will use the API to compare the context properties for testForChangedRC to determine if a context property that you are interested in has changed and then trigger the refresh of the RefreshArea.

For client-side support in invoking a service request, use the declareservice tag. To convert the request properties into JSON object, use json tag.

Basic mechanism to refresh or load a part of the page content

In this section, we discuss the refresh and load events of a page.

wc.widget.RefreshArea

The RefreshArea widget is used to wrap a DOM node that may need to be refreshed by replacing the innerHTML property with fresh HTML loaded from the server. A RefreshArea widget is associated with a registered refresh controller that listens for events that will require this widget to be refreshed.

The parameters of the RefreshArea widget are as follows:

- ▶ **controllerId:** String
This parameter is the ID of the refresh controller to which this widget is registered.
- ▶ **controller:** wc.render.RefreshController
This parameter is the refresh controller instance that is controlling this RefreshArea. This value is initialized by locating the refresh controller with the controller ID specified by controllerId.

- ▶ `objectId`: String

This parameter is a unique ID that can be used to distinguish a `RefreshArea` from other `RefreshAreas` that are controlled by the same refresh controller. If specified, the object ID is passed as a parameter to the URL responsible for providing the refreshed content.

`wc.render.declareRefreshController (initProperties)`

This function declares a new refresh controller and initializes it with the specified initialization properties. The initialization properties are mixed in with the new refresh controller properties.

A refresh controller controls `RefreshArea` widgets. It listens to changes in the render context and changes to the model and decides if the registered `RefreshAreas` should be updated.

The parameters of this function are as follows:

`InitProperties`: The object containing any of these initialization parameters.

- ▶ `id`: String

This parameter is the ID of the refresh controller.

- ▶ `renderContext`: `wc.render.Context`

This parameter is the render context object that is associated with this refresh controller.

- ▶ `url`: String

This parameter is the URL that is called by the refresh function to retrieve the refreshed content for the `RefreshArea` widget.

- ▶ `renderContextChangedHandler`: Function

If defined, this parameter is called when a `renderContextChanged` event is detected. It is called once for each `RefreshArea` widget registered to this refresh controller. The function signature is `(message, widget)`, where `message` is the `renderContextChanged` event message and `widget` is the `RefreshArea` widget.

- ▶ `modelChangedHandler`: Function

If defined, this parameter is called when a `modelChanged` event is detected. It is called once for each `RefreshArea` widget registered to this refresh controller. The function signature is `(message, widget)`, where `message` is the `modelChanged` event message and `widget` is the `RefreshArea` widget.

- ▶ **postRefreshHandler**: Function

If defined, this parameter is called after a successful refresh. The function signature is (widget), where widget is the RefreshArea widget that was just refreshed.

- ▶ **testForChangedRC (propertyNames)**: Function

This parameter looks through the specified array of property names for a render context property that has changed since the last time a renderContextChanged event was detected. This parameter returns true if any of the specified render context properties have changed. Otherwise, it returns false. The function signature is (propertyNames) where propertyNames is an array of property names to be tested.

wc.render.getRefreshControllerByld (id)

This function returns the refresh controller that was declared under the specified identifier. If the refresh controller was not declared, this function will return undefined.

This function has the following parameter, id: String, which is the refresh controller ID.

wc.render.declareContext (id, properties, updateContextURL)

This function declares a new render context and initializes it with the specified render context properties. The update context URL reports changes to the render context to the server.

A render context is a set of client-side context information to keep track off on a page. This context information can be used to decide if changes to RefreshAreas are needed. Changes to the render context can be communicated to the server using an Ajax-style request if an updateContextURL is provided. When the render context is successfully updated, an event is published so that any listening refresh controllers can refresh their content to reflect the updated render context.

This function has the following parameters:

- ▶ **id**: String

This function is the unique ID of the new render context.

- ▶ **properties**: Object

This function is the initial render context properties.

- ▶ **updateContextURL**: String

This function is the URL used to notify the server of render context updates.

wc.render.getContextById (id)

This function gets the render context declared under the specified ID. If the render context was not declared, this function will return undefined.

wc.render.updateContext (id, updates)

This function retrieves the render context with the specified ID, and applies the updates found in the specified updates object.

This function has the following parameters:

- ▶ id: String

This parameter is the render context ID.

- ▶ updates: Object

This parameter is related to render context updates. The properties of the object are added to the render context. Properties that have an undefined value are removed from the render context.

Sample usage

In your page, you can have a RefreshArea <div> for the area you want to load dynamically. The content inside this <div> is refreshed or replaced dynamically with the new content based on the request. Example 3-13 is a simple RefreshArea tag. The most important property is controllerId. This controller is the central point that handles the refresh mechanism for the RefreshAreas registered to it.

Example 3-13 RefreshArea <div> tag for MiniShoppingCart

```
<div dojoType="wc.widget.RefreshArea"
id="MiniShoppingCart"
widgetId="MiniShoppingCart"
controllerId="MiniShoppingCartController">

</div>
```

Example 3-14 shows a simple declaration of RenderContext. This declaration adds client-side render context support to the page. We set the ID for this RenderContext by passing it as a parameter to declareContext method. This ID identifies the render context. If this tag has already been called for this request and ID, then it will do nothing. We kept the RenderContext declarations in the CommonContextsDeclarations.js file.

Example 3-14 RenderContext declaration

```
wc.render.declareContext("ShopCartDisplay_Context",null,"")
```

We include this `CommonContextsDeclarations.js` file in the main JSP, and initialize the `RenderContext` by setting the common properties to that object, as shown in Example 3-15.

Example 3-15 How to include the `RenderContext` declaration in JSP

```
<script type="text/javascript" src="<c:out  
value="\${jspStoreImgDir}javascript/CommonContextsDeclarations.js"/>"></  
script>
```

```
CommonContextsJS.setCommonParameters('${WParam.langId}', '${WParam.sto  
reId}', '${WParam.catalogId}');
```

Set the other parameters (such as local, url, and properties) by using `setContextProperty(contextId,property,value)` method. This method takes `contextId`, `property`, and `value` as the parameters.

Example 3-16 shows how to declare a `RefreshController`, and the properties and handlers that need setting. While declaring `RefreshController` set the ID and the `renderContext` properties. Other optional properties are `url`, which refers to the page the snippet needs to load or reload, and the `formId` that you want to submit. The `RefreshController` declaration has three main handlers:

- ▶ `modelChangedHandler`
This handler is called automatically when business object get updated.
- ▶ `renderContextChangeHandler`
This handler is called when you make a call to load or reload a part of the content in the page just for content change.
- ▶ `postRefreshHandler`
This handler is called when you have the operations needed after the refresh.

Example 3-16 `RefreshController` declaration

```
wc.render.declareRefreshController({  
  id: "MiniShoppingCartController",  
  renderContext: wc.render.getContextById("MiniShoppingCartContext"),  
  url: "",  
  formId: ""  
  
  ,modelChangedHandler: function(message, widget) {  
    var controller = this;  
    var renderContext = this.renderContext;  
    if(message.actionId in order_updated){  
      var param = [];
```

```

        if(message.actionId == 'AjaxAddOrderItem'){
            param.addedOrderItemId = message.orderItemId + "";
            showDropdown = true;
        }
        widget.refresh(param);
    }
    else{
        //alert(message.actionId +" not in order_updated..no
work...");
    }
}

,renderContextChangedHandler: function(message, widget) {
    var controller = this;
    var renderContext = this.renderContext;
}

,postRefreshHandler: function(widget) {
    var controller = this;
    var renderContext = this.renderContext;

    //The dialog contents has changed..so destroy the old dialog with
stale data..
    destroyDialog();

    if(showDropdown){
        //We have added item to cart..So display the drop down with
item added message..

showMiniShopCartDropDown(null,"placeholder",'quick_cart_container','ord
erItemAdded');
        showDropdown = false;
    }
    cursor_clear();
}
}
})

```

We have kept the RefreshController declarations in the commoncontrollerdeclarations.js file. We include this commoncontrollerdeclarations.js file in the main JSP, and set the other properties for the controller as shown in Example 3-17 on page 112.

Example 3-17 CommonControllerDeclaration

```
<script type="text/javascript" src="<c:out  
value="\${jspStoreImgDir}javascript/CommonControllersDeclaration.js"/>">  
</script>
```

```
CommonControllersDeclarationJS.setControllerURL('MiniShoppingCartContro  
ller','\${AjaxQuickCartDisplayURL}');
```

Note: It is essential to follow the order of declarations. RefreshController should always be declared after the RenderContext declaration. It should follow the order because we are using the RenderContext by getting it by ID in RefreshController declaration.

wc.service.declare (initProperties)

This function declares a new Ajax service with the specified ID. A *service* is a server URL that performs a server object create, update, delete or other server processing in WebSphere Commerce. By using this API, you will have the capability to call a WebSphere Commerce server URL using Ajax whenever it is needed in your JavaScript code.

When the service completes successfully, a JSON object containing the response properties of the URL request is returned to the JavaScript functions defined by `successHandle` or `failureHandler`, in case of success or failure respectively.

If the service completes successfully, a `modelChanged` event is sent to subscribed listeners. This means that the following Dojo events are published:

- ▶ `modelChanged`
- ▶ `modelChanged/<actionId>`

This function has the following parameters:

`initProperties`: Object containing any of these initialization parameters.

- ▶ `id`: String
This parameter is the unique ID of this service.
- ▶ `actionId`: String
This parameter is an identifier for the action performed by this service. This value is used in the construction of the topic name for the `modelChanged` event. The `modelChanged` event name is of the form *modelChanged/actionId*.
- ▶ `url`: String
This parameter is the URL for this service.

- ▶ `formId`: String

This parameter is the ID of the form element that is posted to the URL. A service does not necessarily have to be associated with a form element.

- ▶ `validateParameters`: function (parameters)

This parameter validates that the service parameters are correct. This function examines the service parameters to determine if the service is ready to be invoked. The function is called from the `invoke` function before the service is invoked. The default implementation of this function returns *true*. This function may be replaced by passing in a new version with the `initProperties` object when the service is constructed. This parameter returns true if the parameters are valid. It has the parameter of object. This parameter is passed to the `invoke` function.

- ▶ `validateForm`: function (formNode)

This parameter validates that the form values are correct. This function examines the specified form element to determine if the service is ready to be invoked. The function is called from the `invoke` function before the service is invoked. The default implementation of this function returns true. This function may be replaced by passing in a new version with the `initProperties` object when the service is constructed. This parameter returns true if the form values are valid. It has the parameter of element. This parameter has the ID specified by the `formId` property.

- ▶ `successTest`: function (serviceResponse) {

This parameter is the test for a successful service invocation. This function examines the specified service response object to determine if the service was successful or not. The function is called after the response was received from the service. The default implementation will return true if there is no `errorMessage` property in the service response object. This function may be replaced by passing in a new version with the `initProperties` object when the service is constructed. This parameter returns true if the service request is successful. The `serviceResponse` is object. The service response object. This object is the JSON Object returned by the service invocation.

- ▶ `successHandler`: function (serviceResponse)

This parameter performs processing after a successful service invocation. This function is called after a successful service invocation to allow for any post service processing. The default implementation does nothing. This function may be replaced by passing in a new version with the `initProperties` Object when the service is constructed. The `serviceResponse` is Object. The service response object. This object is the JSON Object returned by the service invocation.

- ▶ **failureHandler:** function (serviceResponse)

This parameter performs processing after a failed service invocation. This function is called after a failed service invocation to handle any error processing. The default implementation alerts the user with the error message found in the service response object. This function may be replaced by passing in a new version with the `initProperties` Object when the service is constructed. The `serviceResponse` is Object. The `service` response object. This object is the JSON Object returned by the service invocation.

wc.service.getServiceById (serviceId)

This function gets the service that was declared under the specified identifier. If the service was not declared, this function will return undefined.

This function, illustrated in Example 3-18, is useful if you need to find a service to update some of its parameters before you invoke it.

Example 3-18 Using getServiceById

```
wc.service.getServiceById("AjaxAddOrderItem ").formId =  
"OrderItemForm";
```

Sample usage

The code in Example 3-19 on page 115 shows how to declare a service in JavaScript. We need to set an ID for our service-like ID, `AjaxInterestItemAdd`. When you want to use this service, you can get the service by ID. This ID is a reference to get the service object.

Additional properties are as follows:

- ▶ **url**

This property refers to the service command to which we want to send a request.

- ▶ **actionId**

This property is used by `RefreshControllers` or service handlers to check for the correct action and perform the post operations.

- ▶ **formId**

You can pass a form by using the `formId`. This is useful when somebody wants to add Ajax service to their existing Web 1.0. In this case, they do not need to change much in their forms. They need to add those extra parameters or adjust them according to service requirement and set that `formId` in this property.

Example 3-19 Service declaration using JavaScript

```
wc.service.declare({
    id: "AjaxInterestItemAdd",
    actionId: " AjaxInterestItemAdd",
    url: " AjaxInterestItemAdd",
    formId: "",

    successHandler: function(serviceResponse) {
        alert("success");
    },

    failureHandler: function(serviceResponse) {
        if (serviceResponse.errorMessage) {
            alert(serviceResponse.errorMessage);
        }
    }
});
```

We kept the service declarations in a `ServicesDeclaration.js` file. In the main JSP, we include this `ServicesDeclaration.js` file and set the other properties for the service, as shown in Example 3-20.

Example 3-20 Including the `ServicesDeclaration` and setting common properties.

```
<script type="text/javascript" src="<c:out
value="${jspStoreImgDir}javascript/ServicesDeclaration.js"/>"></script>
<script type="text/javascript">

ServicesDeclarationJS.setCommonParameters('${WCPParam.langId}', '${WCPParam.storeId}', '${WCPParam.catalogId}');
</script>
```

In Example 3-21, we have grouped the services based on the events. These service event mappings are in `ServicesEventMapping.js`. We use this file in the refresh controller to perform the event-based refresh.

Example 3-21 Grouping services based on events

```
<script type="text/javascript" src="<c:out
value="${jspStoreImgDir}javascript/ServicesEventMapping.js"/>"></script>
>
```

Example 3-22 on page 116 show how to invoke the service and how to set the extra parameters before invoking the service. We set the required parameters in a JavaScript array, and pass it to the `invoke` method with `serviceId`.

Example 3-22 Setting parameters and invoking the service

```
var params = [];  
    params.storeId= this.storeId;  
    params.catalogId= this.catalogId;  
    params.langId= this.langId;  
    params.orderId= ".";  
    params.catEntryId= catEntryIdentifier;  
    params.quantity= quantity;  
    wc.service.invoke("AjaxAddOrderItem", params);
```

When the service is invoked it goes to the service command and performs the specified operation. The response comes back to the handler section of the service. If the command succeeds `successHandler` is called. If the command fails `failureHandler` is called. In `failureHandler` you need to put the error handler code. The `ModelChanged` handler in the `RefresherController` will also get called automatically upon response.

The following three examples (Example 3-23 through Example 3-25 on page 117) illustrate how to declare the `RenderContext` and `RefreshController` in JSP code. Considering the best practices of Web 2.0 Store, you should not declare in this way, as this impacts the performance of the store and pages may take a long time to load.

Example 3-23 Declaring a RenderContext in JSP

```
<wcf:declareRenderContext local="true" id="MiniShoppingCartContext" >  
    <jsp:attribute name="properties">{workAreaMode:  
"create"}</jsp:attribute>  
</wcf:declareRenderContext>
```

Example 3-24 Declaring a RefreshController in JSP

```
<wcf:declareRefreshController id="MiniShoppingCartController"  
    url="${AjaxQuickCartDisplayURL}"  
    renderContextId="MiniShoppingCartContext"  
>  
  
    <jsp:attribute name="modelChangedScript">  
        if(message.actionId in order_updated){  
            widget.refresh(renderContext.properties);  
            cursor_clear();  
        }  
    </jsp:attribute>
```

```

    <jsp:attribute name="postRefreshScript">
        //miniShoppingCartJs.quickCartPostRefreshHandler();
    </jsp:attribute>

    <jsp:attribute name="renderContextChangedScript"></jsp:attribute>

</wcf:declareRefreshController>

```

Example 3-25 Declaring a Service in JSP

```

<wcf:declareService id="OrderCopy" actionId="OrderCopy"
url="AjaxOrderCopy" >
    <jsp:attribute name="successHandlerScript">
        for (var prop in serviceResponse) {
            console.debug(prop + "=" + serviceResponse[prop]);
        }

document.location.href="${PrepareOrderURL}&orderId="+serviceResponse.or
derId;
    </jsp:attribute>

    <jsp:attribute name="failureHandlerScript">
        if(serviceResponse){
            for (var prop in serviceResponse) {
                console.debug(prop + "=" + serviceResponse[prop]);
            }
            ErrorHandler.formErrorHandle(serviceResponse, "", "");
        }
    </jsp:attribute>
</wcf:declareService>

```

Attention: We do not suggest using the above way of coding. This is only a possible way of using the RefreshController, RenderContext, and Service. Use this way only when you have limitations using JavaScript and you are not concerned about the performance of your site.

Example 3-26 shows how to create a JSON object. This tag converts the specified Java object to JSON. You can return this JSON object instead returning complete content.

Example 3-26 Declaration of Json tag

```

<wcf:json object="${RequestProperties}">

```

3.2.2 The four scenarios

In this section, we give more explanation with code examples for each of the four scenarios we are talking about. After this section, you will have a broad picture about the scenarios and the flows and you can match these scenarios with any of the page flows in your store, and add these features to your store.

Note: Excerpts from WebSphere Commerce Infocenter were used to provide the following examples. The Infocenter Web page is as follows:

<https://jtcid.hursley.ibm.com/commerce/index.jsp?topic=/com.ibm.commerce.madisons-starterstore.doc/refs/rsmmadisonajaxinteractions.htm>

Scenario 1: Refreshing an area in the page through Ajax request to get the refresh contents

This scenario is for cases where users interact with the UI and areas of the page are refreshed with a lot of new content. This scenario uses the render context, refresh area, and refresh controllers API from the WebSphere Commerce Ajax framework.

A render context object keeps track of client context information. It can trigger `renderContextChanged` events whenever updates occur to any of the properties in the render context object. The refresh controllers are automatically registered to listen to the `renderContextChanged` events. The refresh controller logic determines if the context change should trigger an update of a `RefreshArea` widget. Therefore, in the refresh controller's `renderContextChangedHandler`, use the API to compare the context properties `testForChangedRC` to determine if a context property that you are interested in has changed, and then trigger the refresh of the `RefreshArea`.

Call the `updateContext` method in the `RenderContext` to load or reload the new content in your page. Before making a call to the `updateContext`, set the correct URL for the content snippet in the controller. If you do not set the URL, it takes the default URL, which sets while Controller declaration. When the updater is called it sends a request to the server and gets the `RefreshJSP` snippet and replaces this content in `RefreshArea`. This mechanism remains the same whether you use `DataBeans` or component services.

When using `DataBeans`, the snippet uses `wcbase:useBean` to populate the bean, and gets the data from that bean object. If you are using the component services, the snippet uses `wcf:getData` to get the services object.

As shown in Figure 3-3, the main page sends a request to the server for RefreshJSP. The server compiles the refreshJSP, gets the required data either from a DataBean or from component services, and returns the content to the main page. It is now the RefreshController's responsibility to replace the content in that RefreshArea.

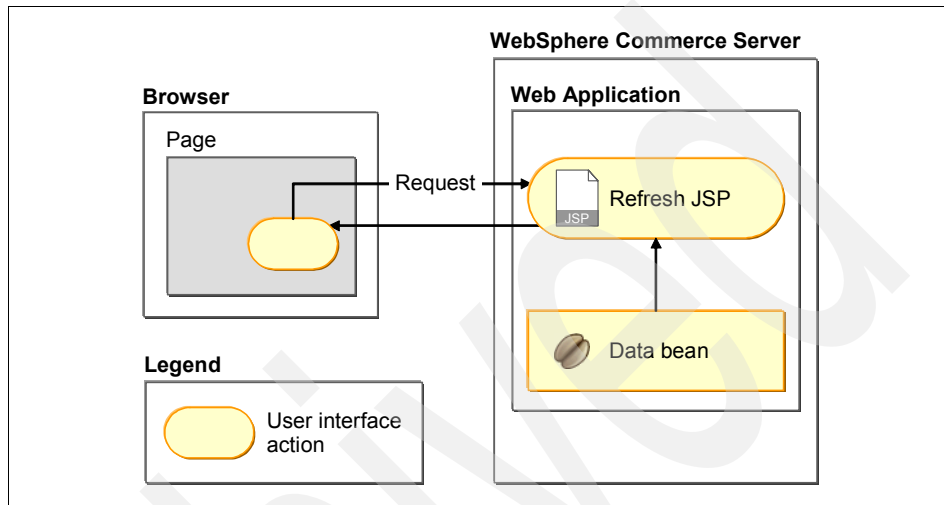


Figure 3-3 How Context change works when the snippet uses DataBean

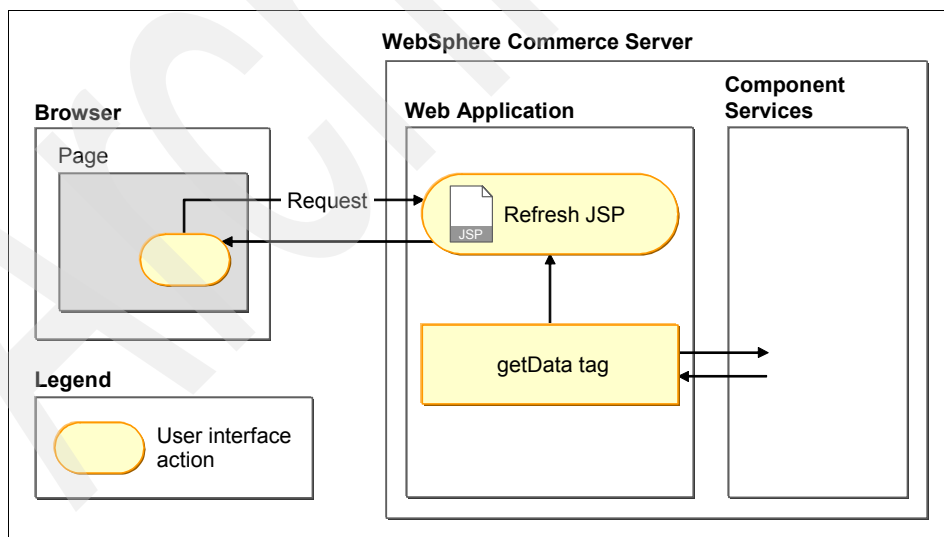


Figure 3-4 How Context change works when the snippet uses component services

This is a common scenario where some new content needs to load or reload in a page. Best example for this scenario is MyAccount page and, pagination in catalog pages, as shown in Figure 3-5. and Figure 3-6 on page 121.

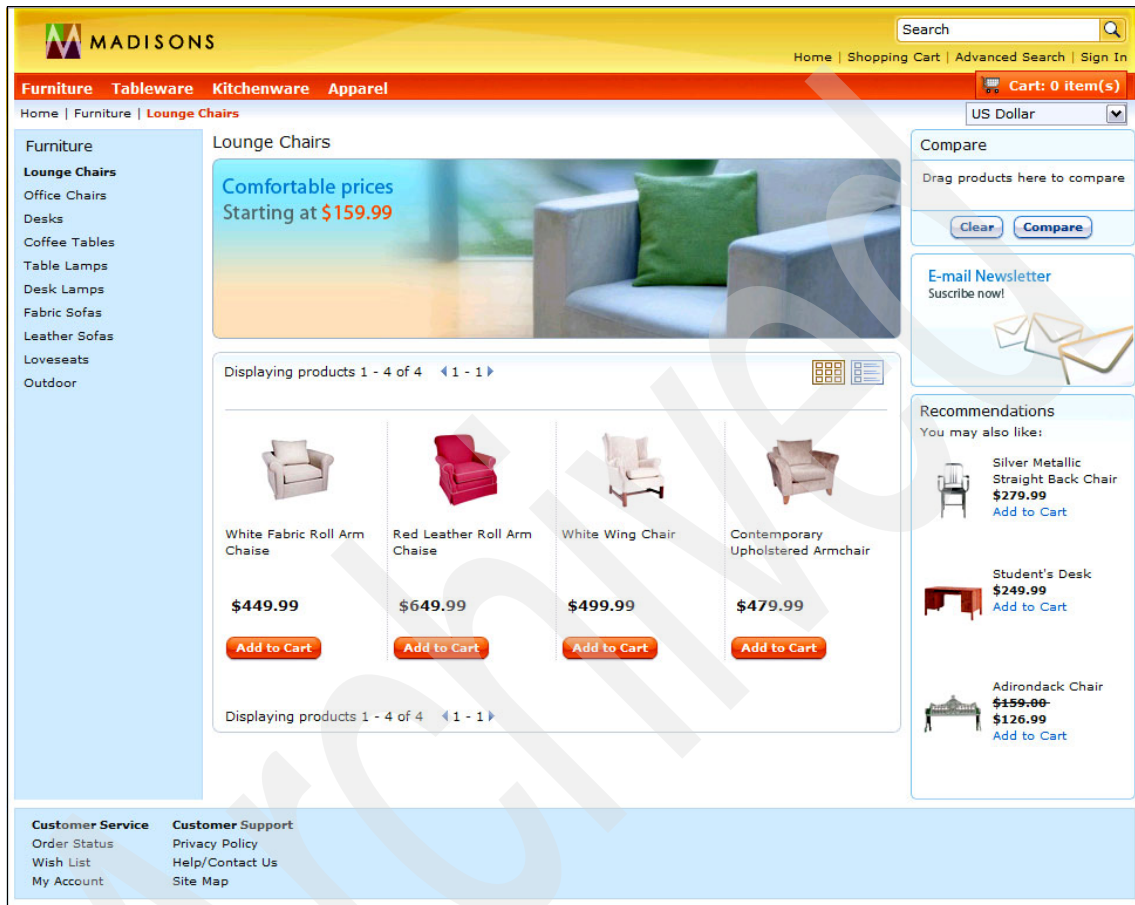


Figure 3-5 Catalog pagination using ContextChange to load the next page content

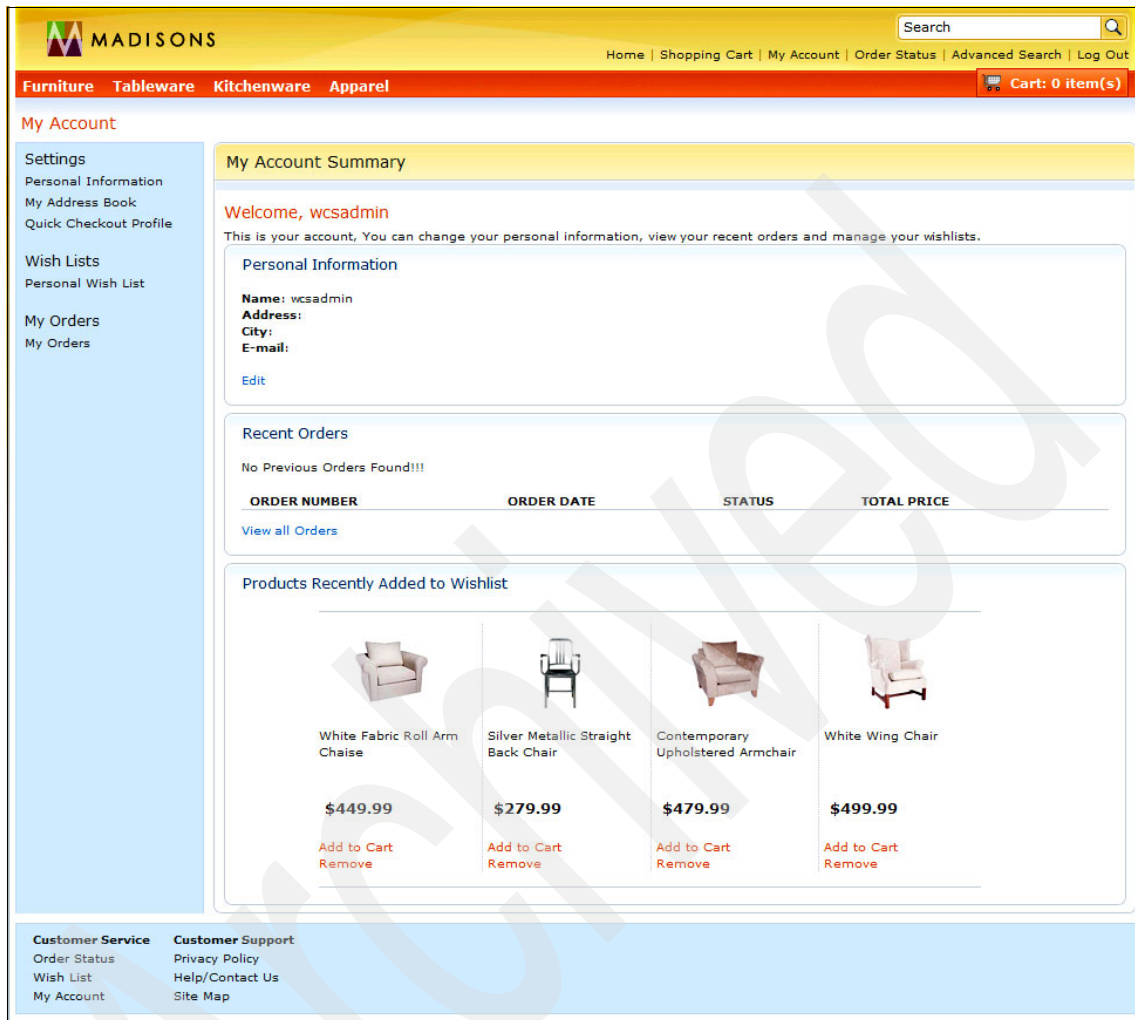


Figure 3-6 MyAccount main page using ContextChange to load the page content

In Figure 3-5 on page 120, the catalog page where we use RefreshArea to load the items in the next page. Figure 3-6 has a main content part in the middle of the page and various links (such as Personal Information, Address Book, Wishlist and so forth) on the left sidebar. When the user clicks any of these links, it loads that particular content in that main content area.

Example 3-27 on page 122 shows a RefreshArea for MyAccount center content. If you want to show some initial content in the RefreshArea you can display that content or import or include the snippet inside the RefreshArea element as shown in Example 3-27 on page 122.

Example 3-27 RefreshArea <div> tag for MyAccount page

```
<div dojoType="wc.widget.RefreshArea"
id="MyAccountCenterLinkDisplay_Widget"
controllerId="MyAccountCenterLinkDisplay_Controller"
role="wairole:region" waistate:live="polite" waistate:atomic="false"
waistate:relevant="all">

    <%out.flush();%>
    <c:import
url="${jspStoreDir}UserArea/AccountSection/MyAccountCenterLinkDisplay.jsp">
        <c:param name="storeId" value="${WCParam.storeId}"/>
        <c:param name="catalogId" value="${WCParam.catalogId}"/>
        <c:param name="langId" value="${langId}"/>
    </c:import>
    <%out.flush();%>
</div>
```

This RefreshArea has a reference to MyAccountCenterLinkDisplay_Controller, which is defined in Example 3-28.

Example 3-28 RefreshController and RenderContext declaration for MyAccount centerlink

```
dojo.require("wc.render.common");

wc.render.declareContext(
    "MyAccountCenterLinkDisplay_Context",
    {workAreaMode: "myAccountMain"},
    "");

wc.render.declareRefreshController({
    id: "MyAccountCenterLinkDisplay_Controller",
    renderContext:
wc.render.getContextById("MyAccountCenterLinkDisplay_Context"),
    url: "",
    formId: ""

    ,modelChangedHandler: function(message, widget) {
        var controller = this;
        var renderContext = this.renderContext;
        console.debug("in model change " + message.actionId);

    }
});
```



```

,renderContextChangedHandler: function(message, widget) {
    var controller = this;
    var renderContext = this.renderContext;
    if (controller.testForChangedRC(["workAreaMode"])) {

        console.debug("test:render context value changed");
        MyAccountDisplay.contextChanged = true; //When context
changed, we need create history tracking information
        widget.refresh(renderContext.properties);
    }
}

,postRefreshHandler: function(widget) {
    //alert("in post refreshhandler");
    var controller = this;
    var renderContext = this.renderContext;
    console.debug("postrefreshscript is invoked");
    ErrorHandler.hideAndClearMessage();

    cursor_clear();
}

});

```

This RefreshController uses MyAccountCenterLinkDisplay_Context. Example 3-29 shows how to set the URL to the controller and call the updateContext for render

Example 3-29 Calling updateContext to update the center content of MyAccount page

```

wc.render.getRefreshControllerById("MyAccountCenterLinkDisplay_Controller").url = resultPageURL;
    wc.render.updateContext("MyAccountCenterLinkDisplay_Context",
{workAreaMode:workAreaModeValue});

```

Scenario 2: Updating business objects in WebSphere Commerce using Ajax, and refreshing multiple areas through subsequent Ajax requests to get the refresh contents

This scenario best fits where the client sends an update request to the server and, upon success, the page needs to be updated again. There are two parts to this scenario:

- ▶ Make an Ajax call to a WebSphere Commerce controller command or WebSphere Commerce service to update a business objects
- ▶ If the above part is successful, make subsequent Ajax get requests to WebSphere Commerce views to retrieve the new HTML contents for each area.

The following two diagrams (Figure 3-7 and Figure 3-8 on page 125) display the interactions that occur between the client and the WebSphere Commerce server in scenario 1.

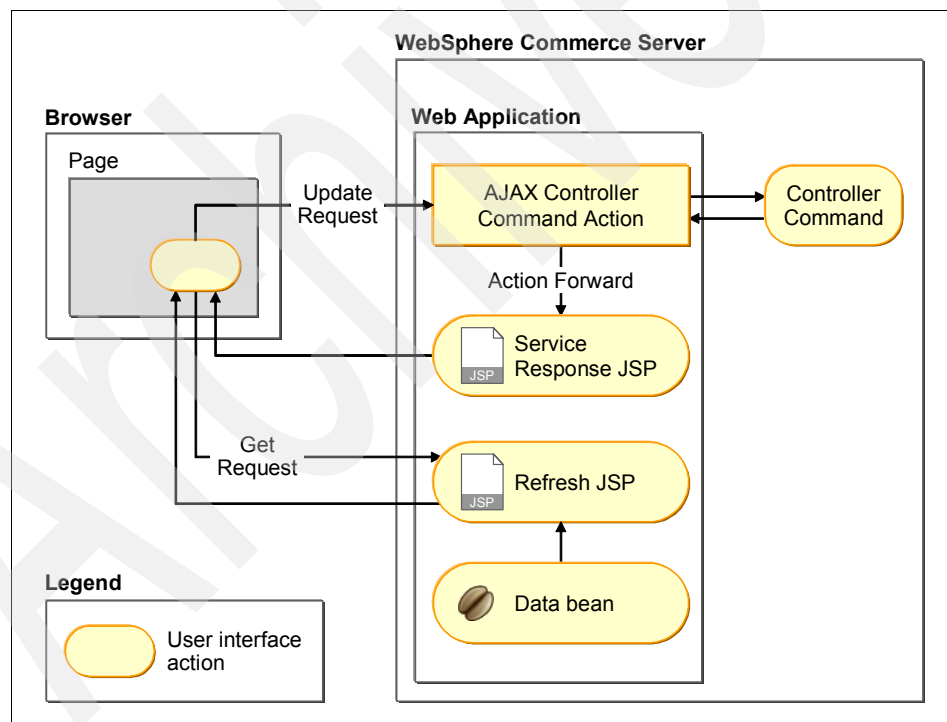


Figure 3-7 Interaction diagram when calling WebSphere Commerce controller commands

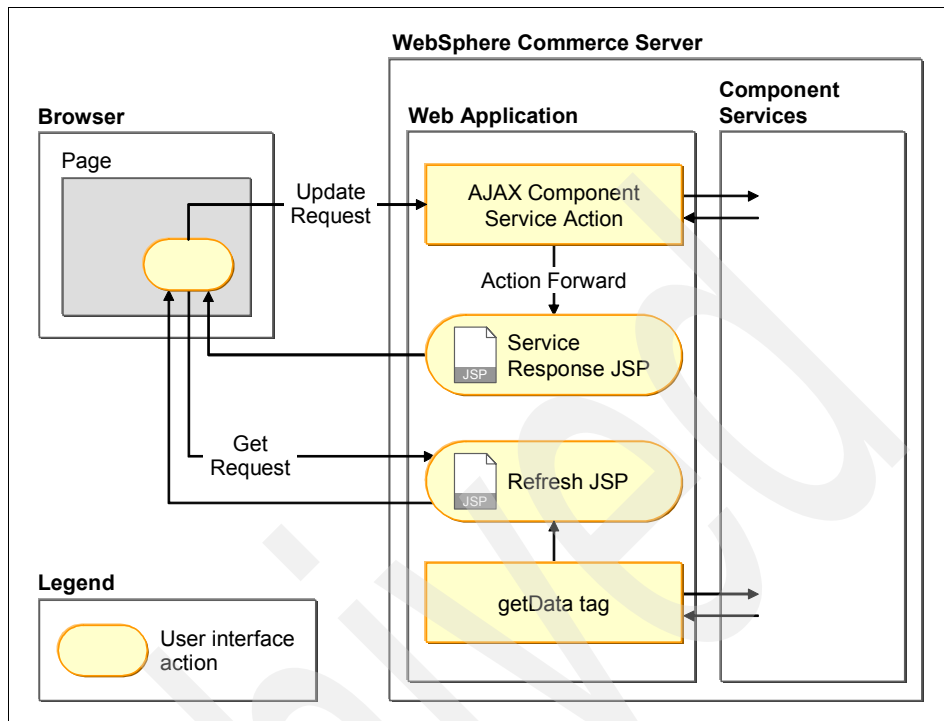


Figure 3-8 Interaction diagram when calling WebSphere Commerce Services

Making an Ajax call to a WebSphere Commerce controller command or service

Before the client is able to call a WebSphere Commerce controller command or service through Ajax, the WebSphere Commerce server must define that the controller command or service can be called using Ajax. This must be done because the regular programming model where you make a request and a redirect to a new is implied by the WebSphere Commerce runtime.

It is simple to define a controller command or service to be available for Ajax-type requests. Define a new struts-action entry in the struts-config XML file that identifies the controller command or service as an Ajax type of action, as shown in Example 3-30 on page 126. For example, to create a new Struts-action for the InterestItemAdd controller command that can be called through Ajax, define this in the Struts configuration XML file.

Example 3-30 Commands

```
<action parameter="order.addOrderItem"
path="/AjaxOrderChangeServiceItemAdd"
type="com.ibm.commerce.struts.AjaxComponentServiceAction">
    <set-property property="authenticate" value="0:0"/>
<set-property property="https" value="0:1"/>
</action>
<action
parameter="com.ibm.commerce.interestitems.commands.InterestItemAddCmd"
path="/AjaxInterestItemAdd"
type="com.ibm.commerce.struts.AjaxAction">
<set-property property="authenticate" value="0:0"/>
<set-property property="https" value="0:1"/>
</action>
```

In an AjaxAction-type URL, after the execution of the command or service, the WebSphere Commerce runtime automatically forwards the request to one of two well-known views to compose a JSON object containing the entries in the response back to the client. The well-known views map to these JSPs, as shown in Example 3-31.

Example 3-31 Well-known view mapping

```
*Runtime
Stores.war/AjaxActionResponse.jsp (success case)
Stores.war/AjaxActionErrorResponse.jsp (failure case)
* WebSphere Commerce Developer
stores/AjaxActionResponse.jsp (success case)
stores/AjaxActionErrorResponse.jsp (failure case)
```

With the Ajax struts-actions defined, you can use the wc.service.declare API in your JavaScript code to define a service for each required URL that you need to call in the page. This wc.service.declare API declares a JavaScript service object that calls WebSphere Commerce Ajax type struts-actions. The service object allows the client to decide when to run the Ajax call. Example 3-32 has the following JavaScript code in the client to define a service object for calling InterestItemAdd through Ajax.

Example 3-32 Service declaration for Ajax OrderItem add

```
wc.service.declare({
    id: "AjaxAddOrderItem",
    actionId: "AjaxAddOrderItem",
    url: "AjaxOrderChangeServiceItemAdd",
    formId: ""
```

```

        ,successHandler: function(serviceResponse) {
            ErrorHandler.hideAndClearMessage();

ErrorHandler.displayStatusMessage(ErrorHandler.errorMessages["SHOPCA
RT_ADDED"]);
        }
        ,failureHandler: function(serviceResponse) {

            if (serviceResponse.errorMessage) {

ErrorHandler.displayErrorMessage(serviceResponse.errorMessage);
            }
            else {
                if (serviceResponse.errorMessageKey) {

ErrorHandler.displayErrorMessage(serviceResponse.errorMessageKey);
                }
            }
            cursor_clear();
        }
    }
}),

```

Example 3-33 AjaxAddOrderItem Service Declaration for shopping cart page

```

wc.service.declare({
    id: "AjaxAddOrderItem_shopCart",
    actionId: "AjaxAddOrderItem",
    url: "AjaxOrderChangeServiceItemAdd",
    formId: ""

    ,successHandler: function(serviceResponse) {
        //Now delete from cart..
        document.location.href =
"AjaxOrderItemDisplayView?storeId=" + ServicesDeclarationJS.storeId
+ "&catalogId=" + ServicesDeclarationJS.catalogId + "&langId=" +
ServicesDeclarationJS.langId;
    }
    ,failureHandler: function(serviceResponse) {

        if (serviceResponse.errorMessage) {

ErrorHandler.displayErrorMessage(serviceResponse.errorMessage);
        }
        else {

```

```

        if (serviceResponse.errorMessageKey) {
            ErrorHandler.displayErrorMessage(serviceResponse.errorMessageKey);
        }
    }
    cursor_clear();
}
}},

```

The above definition in Example 3-32 on page 126 does not trigger the Ajax call to the WebSphere Commerce server. It defines an object. To invoke the Ajax call, use the service invoke API shown in Example 3-34.

Example 3-34 Service invocation API

```

wc.service.invoke("AjaxAddOrderItem");

```

When the request completes successfully, the `successHandler` function defined in the service declaration is executed and the following two `modelChanged` Dojo events are published automatically by the framework:

- ▶ `modelChanged`
- ▶ `modelChanged/<actionId>`

Dojo will automatically notify all subscribed listeners.

Making Ajax calls to WebSphere Commerce views to get refresh contents

The WebSphere Commerce Ajax framework provides two more JavaScript objects, called refresh controllers and refresh areas, to complete this scenario.

You can designate any HTML element in your page as a refresh area. This means that the content within the HTML tag is replaced by new contents from the WebSphere Commerce server whenever the refresh controller associated with the refresh area triggers the refresh request.

Example 3-35 Code defining a refresh area for mini shopping cart

```

<div dojoType="wc.widget.RefreshArea"
    id="MiniShoppingCart"
    widgetId="MiniShoppingCart"
    controllerId="MiniShoppingCartController">

</div>

```

A refresh area always has a refresh controller associated with it. Because refresh controllers are automatically registered to listen to `modelChanged` and `renderContextChanged` events, they are notified when these events occur. These events evaluate the model changes or render context changes and decide if the refresh areas that it manages need to be refreshed. Example 3-36 shows how to define a refresh controller and `renderContext`.

Example 3-36 RefreshController context declarations for mini shopping cart

```
wc.render.declareContext("MiniShoppingCartContext",null,""),

wc.render.declareRefreshController({
  id: "MiniShoppingCartController",
  renderContext: wc.render.getContextById("MiniShoppingCartContext"),
  url: "",
  formId: ""

  ,modelChangedHandler: function(message, widget) {
    var controller = this;
    var renderContext = this.renderContext;
    if(message.actionId in order_updated){
      var param = [];
      if(message.actionId == 'AjaxAddOrderItem'){
        param.addedOrderItemId = message.orderItemId + "";
        showDropdown = true;
      }
      widget.refresh(param);
    }
    else{
      //alert(message.actionId +" not in order_updated..no
work...");
    }
  }

  ,renderContextChangedHandler: function(message, widget) {
    var controller = this;
    var renderContext = this.renderContext;
  }

  ,postRefreshHandler: function(widget) {
    var controller = this;
    var renderContext = this.renderContext;

    //The dialog contents has changed..so destroy the old dialog with
    stale data..
  }
});
```

```

        destroyDialog();

        if(showDropdown){
            //We have added item to cart..So display the drop down with
            item added message..

showMiniShopCartDropDown(null,"placeholder",'quick_cart_container','ord
erItemAdded');
            showDropdown = false;
        }
        cursor_clear();
    }
}},

```

It is important to recognize that the URL defines the WebSphere Commerce server URL to call to get the new HTML contents for the refresh area with which that refresh controller is associated. The `modelChangedHandler` function is performed by the WebSphere Commerce Ajax framework as soon as there is a `modelChanged` event triggered by a successful `wc.service.invoke()` action. The `modelChangedHandler` function checks that the model changed is actually for an `actionId` that pertains to the refresh area. The `widget.refresh()` is the code that makes an Ajax call to the WebSphere Commerce server for the URL specified in the refresh controller. When the new HTML fragment returns from the server, the framework destroys anything that exists in the current refresh area and replaces the old contents with the new HTML fragment returned by the view.

Another useful function is the `postRefreshHandler` function of the refresh controller. This is the last function that the framework calls after a refresh area is successfully refreshed with new contents. It can be used to unblock the UI if it was blocked during the Ajax calls.

Note: The refresh controllers need a render context object. In this scenario, the render context is not used, but you do need to create a dummy render context for each page to make use of refresh controllers and refresh areas.

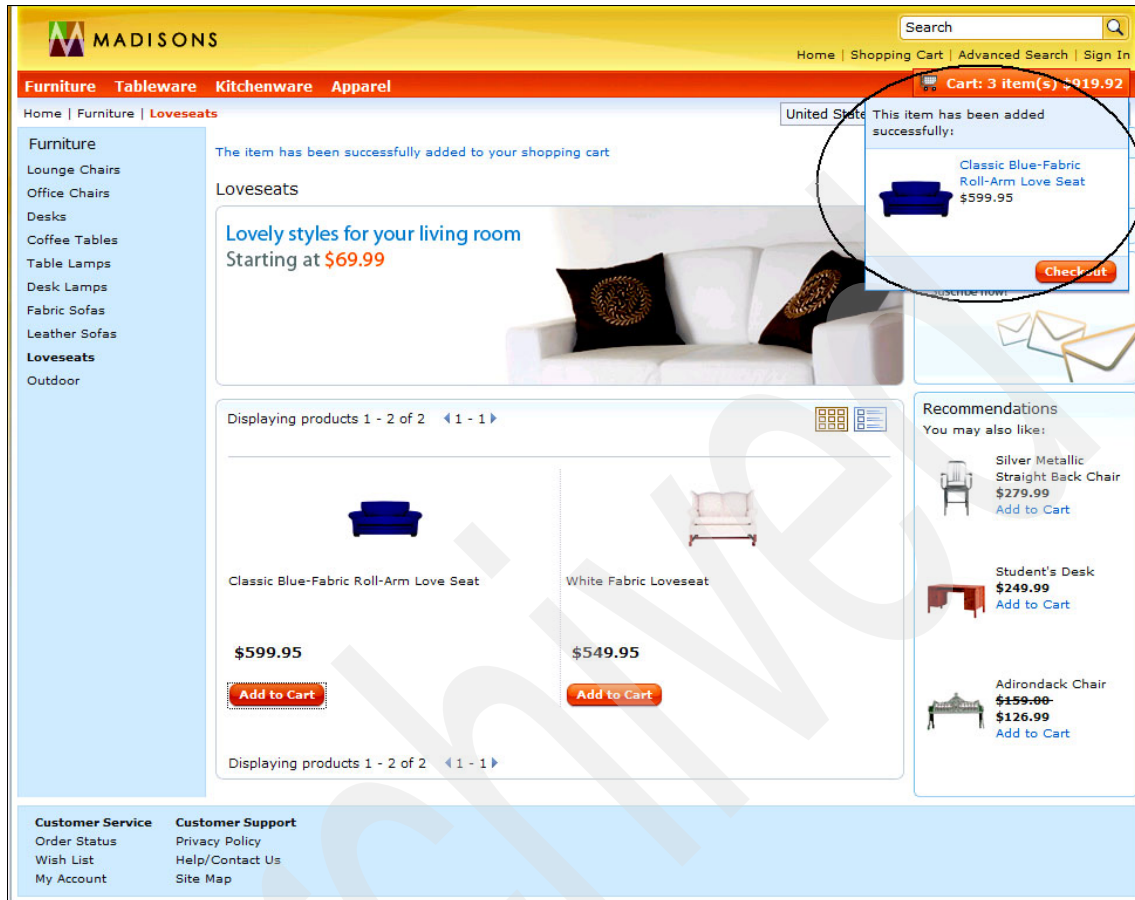


Figure 3-9 Mini shopping cart refresh area

As shown in Figure 3-10 on page 132, adding an item from the catalog page or adding/updating item in Shopping Cart page are the best examples for this scenario.

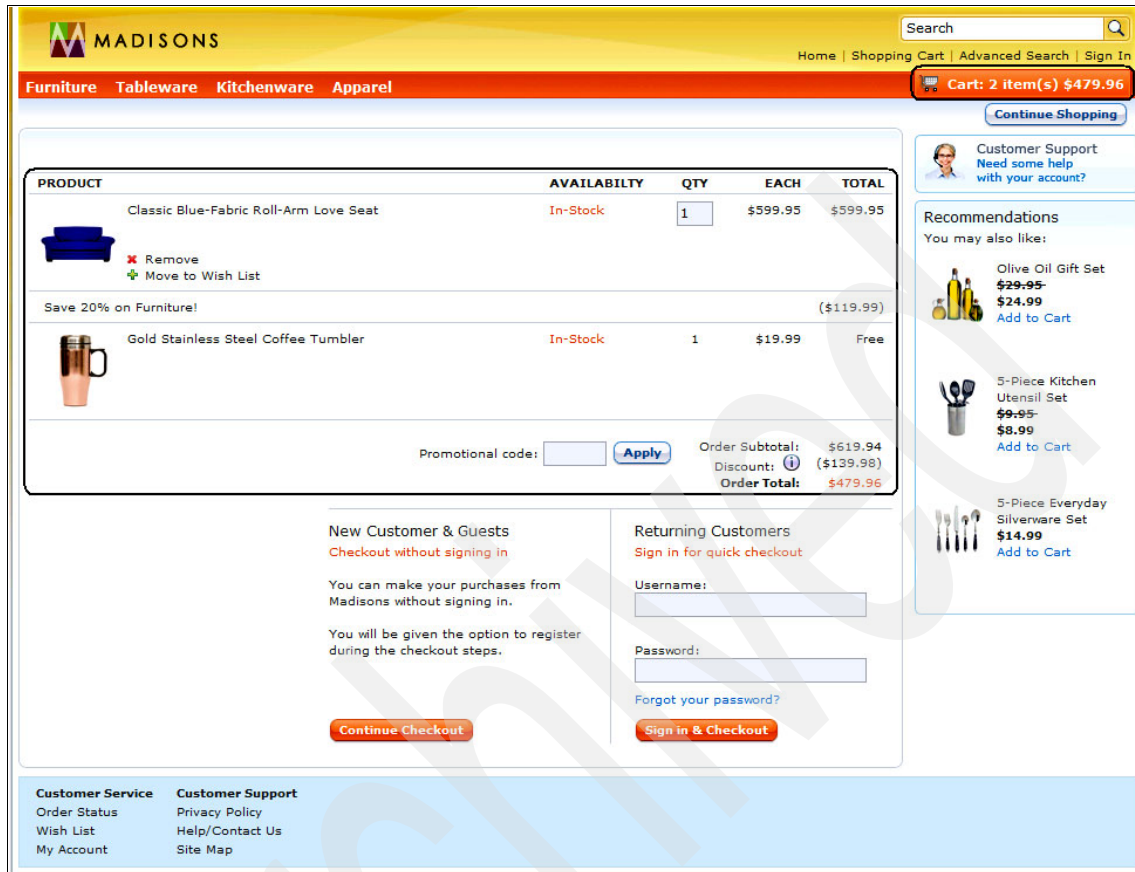


Figure 3-10 Shopping cart page refresh areas

When you add an item from a catalog page, it sends an Ajax service request to the server, as shown in Example 3-34 on page 128. When the response comes back, it goes to the success or failure handler based on whether the request was successful or if it failed. If there are some messages showing the mechanism in each of these handlers as shown in Example 3-32 on page 126 and Example 3-33 on page 127. We have two different service declarations for catalog pages and the shopping cart page, because we are handling the flows differently in each page. Message showing and refreshing mechanisms are different for these pages. The modelChangedHandler handler in the RefreshController is called automatically once the service is successful and response comes. We have conditions which checks for the right actionIds and calls the refresh method.

Scenario 3: Updating business objects in WebSphere Commerce using Ajax, and returning all relevant update information through JSON

This scenario can be achieved using Dojo API directly. There is no need to use the WebSphere Commerce Ajax framework in this scenario. Use the `dojo.xhrPost` API in conjunction with the traditional WebSphere Commerce runtime programming model where a request is made to a controller command or a service, and after execution the WebSphere Commerce runtime redirects the request to whatever is specified in the `URL` or `errorViewName` parameters in the original request. Example 3-37 provides sample code.

Example 3-37 Sending data as a JSON object using `dojo.xhrPost`

```
var parameters = {};  
    parameters.storeId = storeId;  
    parameters.langId=langId;  
    parameters.catalogId=catalogId;  
    parameters.catentryId=productId;  
    parameters.URL="MiniCartContentsJSON";  
parameters.errorViewName="MiniCartContentsJSON";  
  
    dojo.xhrPost({  
        url: "OrderChangeServiceItemAdd",  
        handleAs: "json-comment-filtered",  
        content: parameters,  
        service: this,  
        load: refreshMiniCart,  
        error: function(errObj,ioArgs) {  
            alert("error");  
        }  
    });
```

This code snippet makes an Ajax request to the `OrderChangeServiceItemAdd` service and redirects to the `MiniCartContentsJSON` view, which is mapped to a JSP that creates a JSON object with the desired contents. In our scenario, we assume that the same JSP handles the error scenario as we set the `errorViewName` to the same `MiniCartContentsJSON` view. The client gains control, and either the `load` or `error` function is called depending on success or failure respectively.

Note: An error will only be called when something happens in the communication with the WebSphere Commerce server. Otherwise, a success or exception from WebSphere Commerce code still goes to the load function. It is the load function's job to determine from the JSON object if the request was successful or failed.

The load function uses the JSON object and the DOM manipulation API to replace the elements in the page that need to be updated with new data resulting from the server update.

Note: You do not need to register any of these URLs as AjaxAction. This scenario does not make use of the new WebSphere Commerce Ajax framework. The URLs called in this scenario are defined as before.

Scenario 4: Calling WebSphere Commerce server through Ajax to gather new data as JSON, and refreshing the page contents

This scenario can be achieved using Dojo API directly. There is no need to use the WebSphere Commerce Ajax framework in this scenario. Use the `dojo.xhrPost` API in conjunction with the traditional WebSphere Commerce runtime programming model, where a request is made to a view that maps to a JSP that creates a JSON object. On the load function of the `xhrPost` API, use the DOM manipulation API to put the JSON contents in the Web page elements.

3.3 WebSphere Commerce Framework for Dojo widgets

The WebSphere Commerce Web 2.0 Store uses some Dojo widgets and some extended Dojo widgets to create a versatile, customer-friendly store. These widgets give a Web 2.0 look and feel to your store. This section explains how to use the widgets, and where these widgets best fit in your store. It also provides details how to implement your own widget. The current widgets are highly-customizable, so the section explains how to customize the widgets and use them in several places in the store. After reading this section you will be able to use and customize the existing widgets, and are able to implement new widgets to develop your Web 2.0 Store.

3.3.1 The Dojo widgets used for Web 2.0 Store

In this section we discuss some of the Dojo widgets we are using for Web 2.0 Store, and how they fit in to the store requirement. You will also find examples and sample code for these widgets.

Drag and drop(dojodnd)

Drag and drop, DnD, is a typical feature in a rich client application. It has become one of the most important features in the Web 2.0 Store. The Web 2.0 Store implements the ability to drag items into the shopping cart and compare zone. This section explains how the drag and drop feature works, and how it fits in our store. Section 3.4, “Introduce Web 2.0 features into your Web 1.0 Store” on page 183 shows how to add the drag and drop feature to the ConsumerDirect store to drag products into the mini shopping cart, and how to update the mini shopping cart dynamically after dragging an item.

In our store scenario, we have a source container where the product images and information resides, and a target container where we drag the image from source content. When you drag the item from the source container, check whether it is dragged to the right target. When the item is dropped at the correct target, make the required server requests and handle the response once the server side updating has completed. To declare the source container, use `dojoType=dojo.dnd.Source`, and for the target container. We have listed most relative functions of drag and drop below.

onDndStart: function(source, nodes, copy)

This method is called to initiate the DnD operation. If you want to perform some operations on drag start, you can subscribe those with this method. There are three parameters:

- ▶ **source: Object**
This parameter is the source which provides items.
- ▶ **nodes: Array**
This parameter is the list of transferred items.
- ▶ **copy: Boolean**
This parameter copies items if it is set to true. This parameter move items otherwise.

onDndSourceOver: function(source)

This method is called when detected a current source. you can use this to for proper source check. This method takes one parameter, source: Object. This parameter is the source which has the mouse over it.

onDndDrop: function(source, nodes, copy)

This method is called to finish the DnD operation. You can subscribe to a set of statements or a method to this ondrop event. This method has three parameters:

- ▶ **source:** Object
This parameter is the source which provides items
- ▶ **nodes:** Array
This parameter is the list of transferred items
- ▶ **copy:** Boolean
This parameter copies items if set to true. This parameter moves items otherwise.

destroy: function()

This method is called internally to prepare the object to be garbage-collected. You can call this externally as well, when you are operating with multiple DnD objects.

copyState: function(keyPressed)

This method will check the type of DnD operation for the current object. This method copies items if set to true. This method moves items otherwise. It has one parameter, **keyPressed:** Boolean, indicating the copy was pressed.

checkAcceptance: function(source, nodes)

This method checks the acceptance of the target with current drag item. It has two parameters:

- ▶ **source:** Object
This parameter is the source which provides items.
- ▶ **nodes:** Array
This parameter is the list of transferred items.

Note: This method is called internally when you drop an item. It checks the existence of `dragsourcetype` in the acceptance types of the target and returns the boolean.

deleteSelectedNodes: function()

This method deletes all selected items. This is useful when you want to perform your operation on drop of items.

Sample usage of dojo.dnd.source

Example 3-38 shows how to use `dojo.dnd.source` in your code, and how to put the drag items in this source. The main parameters set to the drag source are as follows:

- ▶ `id`
This parameter indicates a unique ID to differentiate this with other drag sources.
- ▶ `jsId`
- ▶ `dndType`
This is the drag source type. It is checked with acceptance types when the item drops.
- ▶ `copyonly`
This parameter is flagged to indicate that the item should not be moved during drag because it is a copy.

Example 3-38 Using `dojo.dnd.source`

```
<div dojoType="dojo.dnd.Source" jsId="dndSource"
id="${catEntryIdentifier}" copyOnly="true" dndType="${dragType}">
  <div class="dojoDndItem" dndType="${dragType}">
    <c:choose>
      <c:when test="${!empty catEntryDescription.thumbnail}">
        <a href="<c:out value="${catEntryDisplayUrl}"
escapeXml="false"/>" id="img<c:out value="${catEntryIdentifier}" />"
class="itemhover"
onfocus="showPopupButton('${prefix}_${catEntryIdentifier}');">
          "
alt="<c:out
value="${fn:replace(catEntryDescription.name, search, replaceStr)}
${displayPriceString}" escapeXml="false"/>"
border="0"/>
        </a>
      </c:when>
      <c:otherwise>
        <a href="<c:out value="${catEntryDisplayUrl}"
escapeXml="false"/>" id="img<c:out value="${catEntryIdentifier}" />"
class="itemhover"
onfocus="showPopupButton('${prefix}_${catEntryIdentifier}');">
          <c:out
value="${jspStoreImgDir}" />images/NoImageIcon_sm.jpg"
```

```

        alt="<c:out
value="\${fn:replace(catEntryDescription.name, search, replaceStr)}
\${displayPriceString}" escapeXml="false"/>"
        border="0"/>
    </a>
</c:otherwise>
</c:choose>
</div>
</div>

```

Example 3-39 shows how to use `dojo.dnd.Target`. The main parameters are as follows:

- ▶ `id`
This parameter indicates a unique ID.
- ▶ `accept`
This parameter accepts the types that the drag target can accept.
- ▶ `jsId`

A target can have any number of accept types, so the target can accept multiple drag item types from any number of drag sources. Similarly, the items in the drag source can be dragged to any target available, but the target only accepts the acceptable item types.

Example 3-39 Using `dojo.dnd.Target`

```

<div dojoType="dojo.dnd.Target" jsId="miniShopCart_dndTarget"
id="miniShopCart_dndTarget" accept="item, product, package, bundle" >
    <div id="shopping-cart">
        <div id="shopcartContainer">
            <div dojoType="wc.widget.RefreshArea" id="MiniShoppingCart"
widgetId="MiniShoppingCart" controllerId="MiniShoppingCartController"
onmouseover="showMiniShopCartDropDown(event, 'placeholder', 'quick_cart_c
ontainer', 'orderItemsList');" role="wairole:region"
waistate:live="polite" waistate:atomic="true" waistate:relevant="all">
                <%out.flush();%>
                <c:import
url="\${jspStoreDir}include/MiniShopCartDisplay.jsp">
                    <c:param name="storeId" value="\${WCParam.storeId}"/>
                    <c:param name="catalogId"
value="\${WCParam.catalogId}"/>
                    <c:param name="langId" value="\${langId}"/>
                </c:import>
                <%out.flush();%>
            </div>
        </div>
    </div>

```



```

        </div>
    </div>
</div>
</div>

```

There are different kinds of handling methods to handle the process after an item is dropped in the target. We show the method which best fits in to our Web 2.0 Store requirement. It does not mean that the operation is successful once the acceptable item is dropped in the target area. In fact, the actual process starts at this point. The next paragraph explains the handling of the process once the item drops to the mini shopping cart target zone, as shown in Figure 3-11.

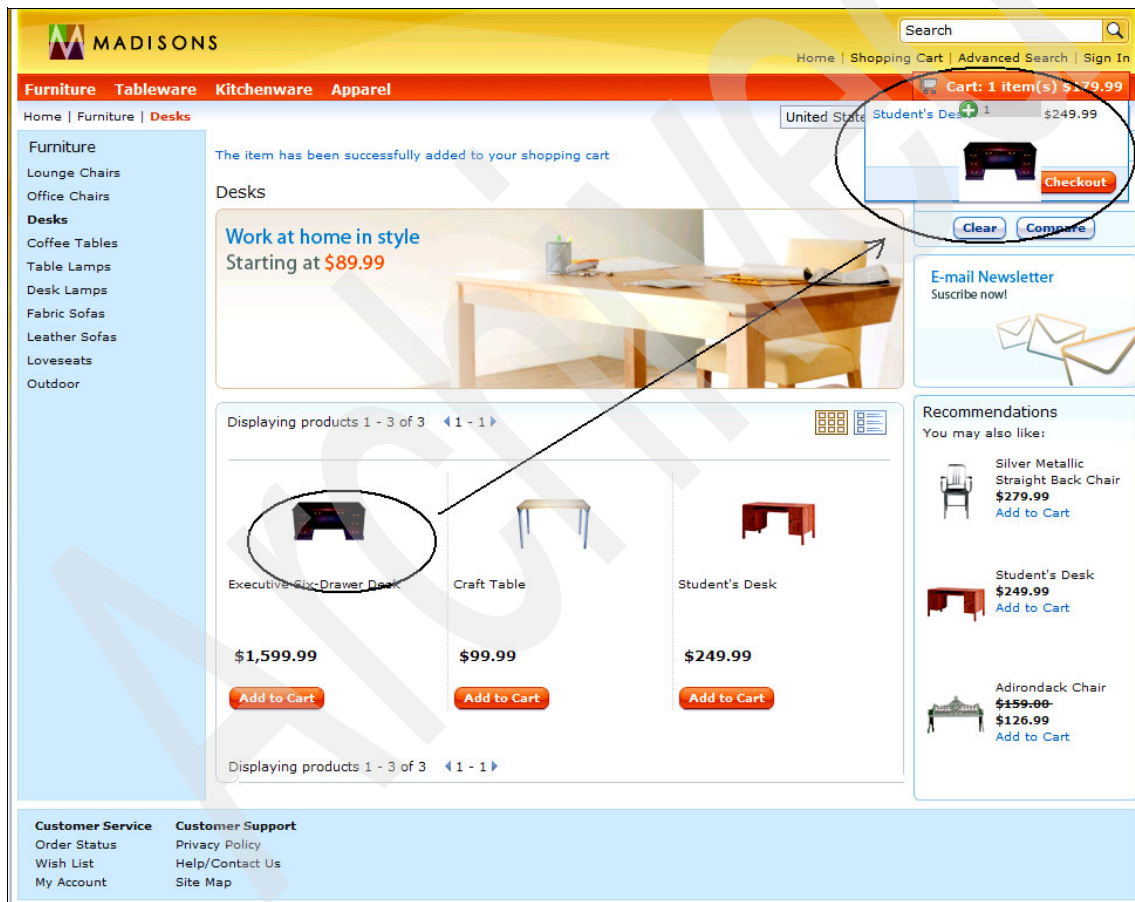


Figure 3-11 Dragging to the shopping cart

When the catalog item drops to mini shopping cart drop zone, it calls the set of statements shown in Example 3-40, because we have subscribed this with the DnD method shown in first line of Example 3-40. This subscribe process should be a part of page load, so you can put these statements in `dojo.onload()`.

Example 3-40 Handling an ondrop event

```
dojo.subscribe("/dnd/drop", function(source, nodes, copy, target){

    target.deleteSelectedNodes();

    if(target.parent.id=='miniShopCart_dndTarget'){
        if(source.node.getAttribute('dndType')== 'item' ||
source.node.getAttribute('dndType')== 'package') {
            categoryDisplayJS.AddItem2ShopCartAjax(source.parent.id
,1);
        } else if(source.node.getAttribute('dndType')== 'product' ||
source.node.getAttribute('dndType')== 'bundle') {

showPopup(source.parent.id, '${WCPParam.storeId}', '${WCPParam.langId}', '${
WCPParam.catalogId}', function(e){return e;}, 'miniShopCart_dndTarget');

        }
    }
});
```

The target subscribed function has the following parameters:

- ▶ source (drag source)
- ▶ nodes (drop nodes)
- ▶ copy

Setting this parameter to true will move the item. Set this parameter to false and the item will not move.

- ▶ target (drop target)

When the item is dropped to the mini shopping cart, it will add the item image to the target. We are not going to show the image as is, because the actual process starts at this point. When the process completes successfully, we update the content of the target(MiniShoppingCart) using RefreshArea.

In the first statement in the target method, we delete the `selectedNodes`. We then check whether the item dropped at the right target (the mini shopping cart). If the item dropped at the right target, we check for the `dndType`. There are different processes for different `dndTypes`. For example, if the `dndType` is `item/package`, we call for further processing, which adds an item to the cart. If the `dndType` is

bundle or product, we call showPopup, which displays a pop-up window with more details about the product, and allow user to do more selection.

In our store, we use the DnD feature in two places. We drag items to the cart and we drag items to the compare zone as shown in Figure 3-11 on page 139 and Figure 3-12. We only explained one flow. The other flow is similar.



Figure 3-12 Dragging to the compare zone

Tooltip(dijit.Tooltip)

We are using Dojo tooltip to show the alerts and error messages in the store. We have slightly modified the way the tooltip behaves to match our requirements. Instead of showing a tooltip on when hovering the mouse over it, we use it to show an invalid input entry or an error message. The typical scenario for tooltip is user entries in the UserRegistration page. The following list details some important tooltip properties and methods:

- ▶ **label:** (String)
This is the text to display in the tooltip. Specified as innerHTML when creating the widget from markup.
- ▶ **showDelay:** (Integer)
This is the number of milliseconds to wait after hovering over/focusing on the object, before the tooltip is displayed.
- ▶ **connectId:** (String[])
This is the ID (or IDs) of domNodes to which you attach the tooltip. When the user hovers over any of the specified DOM nodes, the tooltip displays.

open: function(/*DomNode*/ target)

This method will display the tooltip. You can pass a target node where you want to display this tooltip. It has one parameter, target, which indicates the DomNode where you want to display this tooltip.

close: function(){

This method hides the tooltip. This is a counter for open but this is not called directly.

dijit.showTooltip = function(innerHTML,aroundNode)

This method is used to display tooltip with specified contents for a specified node. It has two parameters:

- ▶ **innerHTML:** (String)
This parameter indicates the HTML content that you want to display in the tooltip.
- ▶ **aroundNode**
This parameter indicates the node where you want to display the tooltip.

dijit.hideTooltip = function(aroundNode)

This method will hide the tooltip for a specified node. It has one parameter, aroundNode, which indicates the node for which you want hide the tooltip.

Sample usage of tooltip

Example 3-41 on page 143 illustrates using the tooltip to show the error messages in a store. In any page, when an error occurs due to wrong user input, the formErrorHandleClient method is called. In turn, this method will show the tooltip with the error message content against the node, as shown in Figure 3-13 on page 144.

```
formErrorHandleClient:function(id,errorMessage){
    // summary: This function will show the an error message tooltip
    // around the input field with the problem.
    // Description: This function will check for the emptiness of the
    required filed and displays the
    // "errorMessage" related to that field as a tooltip. The tooltip
    will be closed on focus lost.
    // id: The identifier for the filed in the form.
    // errorMessage : The message that should be displayed to the user.

    var element = dojo.byId(id);
    if(element){
        if (this.identifier != (id + "_tooltip")) {
            this.identifier = id + "_tooltip";
            var node = document.createElement('span');
            node.innerHTML = errorMessage;
            var tooltip = new dijit.Tooltip({connectId: [id]}, node);
            tooltip.startup();
            console.log("created", tooltip, tooltip.id);
            element.focus();
            tooltip.open(element); // force to have this for IE if the
error is on a link (i.e. <a>)
            dojo.connect(element, "onblur", tooltip, "close"); // force
to have this for IE if the error is on a link (i.e. <a>)
            dojo.connect(element, "onblur", tooltip, "destroy");
            dojo.connect(element, "onblur", this,
"clearCurrentIdentifier");
        }
    }
}
```

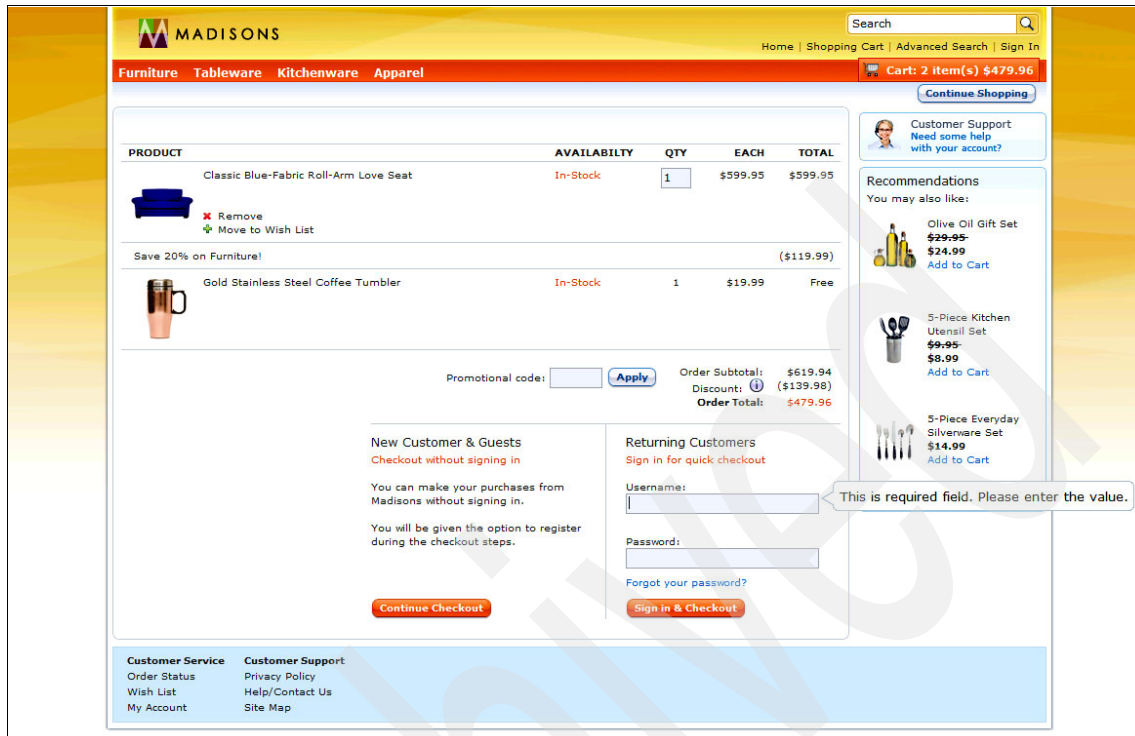


Figure 3-13 Tooltip used as error message bubble

Dialog (dijit.Dialog)

We are using the Dojo dialog to show the ProductQuickInfo in the store, as shown in Figure 3-14 on page 145. ProductQuickInfo is a pop-up window that displays extra information about the product.

Upon hovering the mouse over a product image, a **quickinfo** button displays. Clicking the button shows the product information pop-up window for that product. For this scenario, the dijit.Dialog fits exactly as per the requirement. Important methods in dijit.Dialog are described below.

show: function()

This method displays the dialog box for the current object.

hide: function()

This method hides the dialog box for the current object.

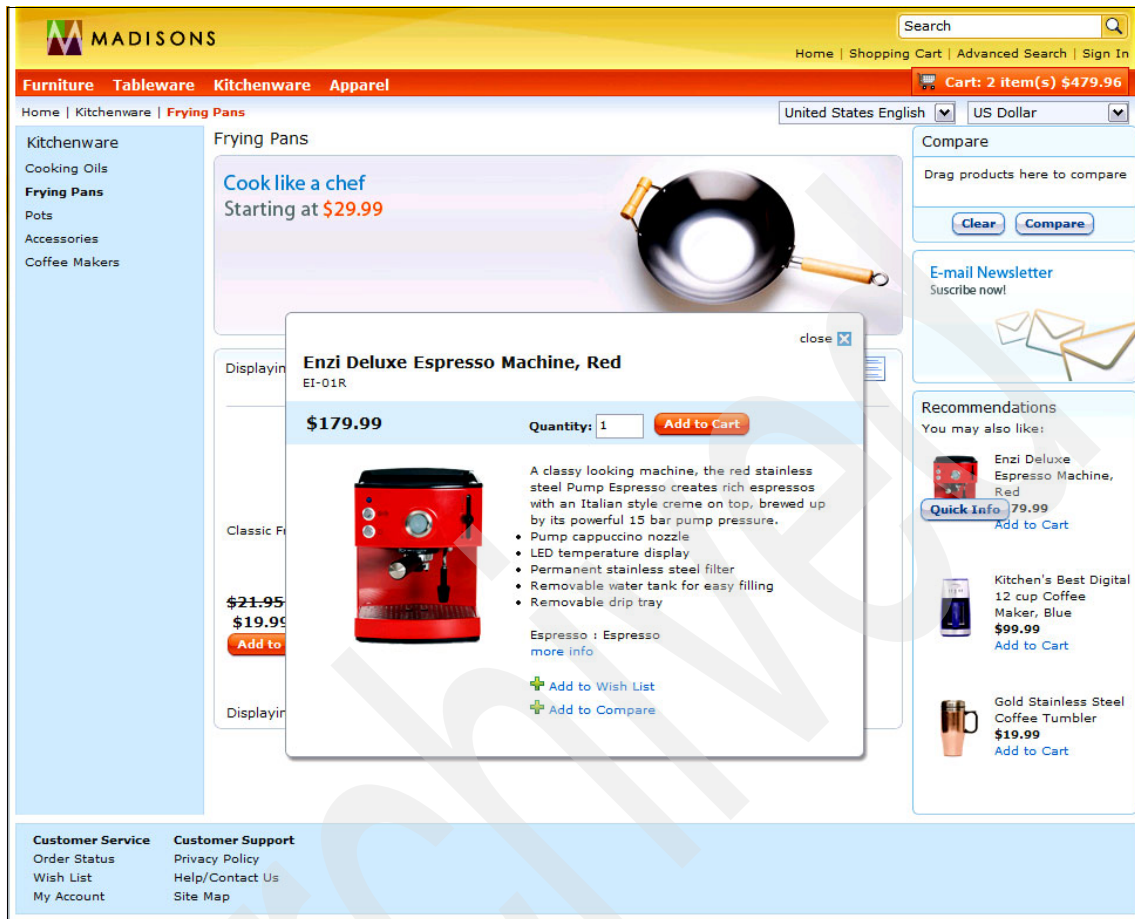


Figure 3-14 Product Quickinfo popup

Sample usage of dialog

As per our store scenario, hovering the mouse over any product will display the **quickinfo** button. Clicking that button displays a quickinfo pop-up window. Example 3-42 on page 146 shows to hide the **quickinfo** button.

Note: The **quickinfo** button is just one option to show quickinfo pop-up window. If you want to display the pop-up window on a different action, like onmouseover on the product itself you can show the popup. In this case, this step can be skipped, and you can have your own flow to show popup.

*Example 3-42 Displaying and hiding the **quickinfo** button for an item*

```
function showPopupButton(id){
    var popupButtonStyle =
document.getElementById("popupButton_"+id).style;
    popupButtonStyle.visibility="visible";

}

function hidePopupButton(id){
    var popupButtonStyle =
document.getElementById("popupButton_"+id).style;
    popupButtonStyle.visibility="hidden";
}
```

Example 3-43 shows sample content for a quickinfo pop-up window. The content of the quickinfo pop-up window is enclosed in a dialog <div> tag. This is the parent element for the content. The pop-up window contains the title, a close link, and the body of the content. The close link is used to hide the pop-up window. In the body, set the UI without filling in the data. It is loaded dynamically once the pop-up window is called.

Example 3-43 Sample product quickinfo area

```
<div id="second_level_category_popup" dojoType="dijit.Dialog"
onkeypress="javascript:hidePopup('second_level_category_popup',event);"
>
    <div class="top_left">images/trasparent.gif" /></div>
    <div class="top_right">images/trasparent.gif" /></div>
    <div class="header" id="popupHeader">
        <div class="close"><a id="closeLink"
href="javascript:hidePopup('second_level_category_popup');"
onkeydown="javascript:setbackFocus(event);"><div class="icon">popup_close.png"
/></div><fmt:message key="QuickInfo_Close"
bundle="\${storeText}"/></a></div>
    </div>

    <div class="bodyarea">
        <div class="bodycontent">
            <div class="title">

        </div>
```



```

        <div class="purchase_details">

            </div>
        </div>
    </div>

    <div class="footer">
        <div class="bot_left"></div>
        <div class="bot_right"></div>
    </div>
</div>

```

Clicking **quickinfo** calls the showPopup method with the following parameters:

- ▶ productId
This is the product for which the pop-up window will display information.
- ▶ storeId
- ▶ LangId
- ▶ catalogId
- ▶ event
This parameter can be a key press or mouse click event.
- ▶ targetId
- ▶ nodeId

Example 3-44 shows the sample code for the showPopup method.

Example 3-44 showPopup method sample code

```

function showPopup(productId,storeId,langId, catalogId,
event,targetId,nodeId,productActionList){
    if(event == null || (event != null && event.type!="keypress") ||
(event != null && event.type=="keypress" && event.keyCode==13)){
        resetPopUp();

        // All the code related to ui content goes here

        digit.byId('second_level_category_popup').closeButtonNode.style.display
        ='none';
        digit.byId('second_level_category_popup').show();
    }
}

```

```

        // hides the DialogUnderlayWrapper component, the component that
        // grays out the screen behind,
        // as we do not want the background to be greyed out
        dojo.query('.dijitDialogUnderlayWrapper',
        document).forEach(function(tag) {
            tag.style.display='none';
        });

        var parameters = {};
        parameters.storeId = storeId;
        parameters.langId=langId;
        parameters.catalogId=catalogId;
        parameters.productId=productId;

        dojo.xhrPost({
            url: "GetCatalogEntryDetailsByID",
            handleAs: "json-comment-filtered",
            content: parameters,
            service: this,
            load: populatePopUp,
            error: function(errObj,ioArgs) {
                alert("error");
            }
        });
    }
}

```

The flow is as follows:

1. The showPopup method sets the required styles.
2. The showPopup method calls the resetPopup method. The resetPopup method resets the data if there is an existing pop-up window.
3. The show method is called, which opens the pop-up window. The window is a simple UI without any data.
4. The next statement makes an Ajax request to the server to get data for the pop-up window. It sets populatePopUp as the call back method.
5. When the server responds, populatePopUp is called automatically.
6. The populatePopUp method sets the data it got from the server to the pop-up menu user interface. Example 3-45 on page 149 shows the sample code for the populatePopUp method. This method retrieves the userinterface elements by ID and sets the data to them, even the content for the actions in the pop-up menu also handled by this method.

```
function populatePopUp(serviceResponse, ioArgs) {
    var catEntryID =
serviceResponse.catalogEntry.catalogEntryIdentifier.uniqueID;
    var isProductBean = false;
    if(serviceResponse.catalogEntry.catalogEntryTypeCode=='ProductBean') {
        isProductBean = true;
    }
    document.getElementById('productName').innerHTML =
serviceResponse.catalogEntry.description[0].name;
    document.getElementById('productFullImage').src =
serviceResponse.catalogEntry.description[0].fullImage;
    document.getElementById('productLongDescription').innerHTML =
serviceResponse.catalogEntry.description[0].longDescription + '<br />';

    document.getElementById('productSKUValue').innerHTML =
serviceResponse.catalogEntry.catalogEntryIdentifier.externalIdentifier.pa
rtNumber;
    document.getElementById('productMoreInfoLink').href =
serviceResponse.catalogEntryURL;

    for (var i in serviceResponse.catalogEntryPromotions) {
        document.getElementById('productPromotions').innerHTML =
serviceResponse.catalogEntryPromotions[i] + '<br />';
    }

    var addToCart;
    if(document.getElementById('addToCartLinkAjax')){
        if(isProductBean) {
            addToCart = document.getElementById('addToCartLinkAjax');
            addToCart.href =
"JavaScript:categoryDisplayJS.Add2ShopCartAjax('entitledItem_"+catEntryID
+"',document.getElementById('productPopUpQty').value);
hidePopup('second_level_category_popup');"
        } else {
            addToCart = document.getElementById('addToCartLinkAjax');
            addToCart.href =
"JavaScript:categoryDisplayJS.AddItem2ShopCartAjax('"+catEntryID+"',docum
ent.getElementById('productPopUpQty').value);
hidePopup('second_level_category_popup');"
        }
    }
    if(document.getElementById('addToCartLink')){
        if(isProductBean) {
```

```

        addToCart = document.getElementById('addToCartLink');
        addToCart.href =
"JavaScript:categoryDisplayJS.Add2ShopCart('entitledItem_"+catEntryID+"',
document.getElementById('OrderItemAddForm_"+catEntryID+"'),document.getEl
ementById('productPopUpQty').value);
hidePopup('second_level_category_popup');"
    } else {
        addToCart = document.getElementById('addToCartLink');
        addToCart.href =
"JavaScript:categoryDisplayJS.AddItem2ShopCart(document.getElementById('O
rderItemAddForm_"+catEntryID+"'),document.getElementById('productPopUpQty
').value); hidePopup('second_level_category_popup');"
    }
}

if (document.getElementById('addToWishListLinkAjax')) {
    document.getElementById('addToWishListLinkAjax').href =
"JavaScript:categoryDisplayJS.Add2WishListAjax('entitledItem_"+catEntryID
+"'); hidePopup('second_level_category_popup');"
} else if (document.getElementById('addToWishListLink')) {
    document.getElementById('addToWishListLink').href =
"JavaScript:categoryDisplayJS.Add2WishList('entitledItem_"+catEntryID+"',
document.getElementById('OrderItemAddForm_"+catEntryID+"'));
hidePopup('second_level_category_popup');"
}
if (document.getElementById('addToCompareLink')) {
    document.getElementById('addToCompareLink').href =
"JavaScript:categoryDisplayJS.Add2CompareAjax('"+catEntryID+"', '" +
serviceResponse.productCompareImagePath +"', '" +
serviceResponse.catalogEntryURL+ "')"; delayHidePopup();"
}
if (document.getElementById('replaceCartItemAjax')) {
    document.getElementById('replaceCartItemAjax').href =
"JavaScript:categoryDisplayJS.ReplaceItemAjax('entitledItem_"+catEntryID+
"',document.getElementById('productPopUpQty').value);
hidePopup('second_level_category_popup');"
}
if (document.getElementById('replaceCartItemNonAjax')) {
    document.getElementById('replaceCartItemNonAjax').href =
"JavaScript:categoryDisplayJS.ReplaceItemNonAjax('entitledItem_"+catEntry
ID+"',document.getElementById('productPopUpQty').value,document.getElemen
tById('ReplaceItemForm')); hidePopup('second_level_category_popup');"
}
}
}

```

Example 3-46 shows how to handle the hide pop-up window. To hide the pop-up window we make the display = none for the element instead of destroying the pop-up window. This improves the performance by keeping the same instance of the pop-up window and refreshing the content for different products.

Example 3-46 Hiding the pop-up window

```
function hidePopup(id,event){
if(event!=null && event.type=="keypress" && event.keyCode!="27"){
    return;
}else{
    var popupViewStyle = document.getElementById(id).style;
    popupViewStyle.display="none";
}
}
```

When we click **quickInfo**, a pop-up window displays. It will try to load the data by making a server call. If there is already an instance of a pop-up window, it might show information about some other product. Making a call to the resetPopUp method clears this old data. This method is called from showPopUp, as shown in Example 3-47.

Example 3-47 Resetting the data content in the pop-up window

```
function resetPopUp() {
    document.getElementById('productName').innerHTML = "";
    document.getElementById('productPrice').innerHTML = "";
    document.getElementById('productLongDescription').innerHTML = "";
    document.getElementById('productDescriptiveAttributes').innerHTML
= "";
    document.getElementById('productSKUValue').innerHTML = "";
    document.getElementById('productMoreInfoLink').href = "";
    document.getElementById('productPromotions').innerHTML = "";
    if(document.getElementById('productPopUpQty')){
        document.getElementById('productPopUpQty').disabled = false;
        document.getElementById('productPopUpQty').value = "1";
    }

    document.getElementById('productAttributes').innerHTML = "";
    if (document.getElementById('addToCartLinkAjax')) {
        document.getElementById('addToCartLinkAjax').href = "";
    } else if (document.getElementById('addToCartLink')) {
        document.getElementById('addToCartLink').href = "";
    }

    if (document.getElementById('addToWishListLinkAjax')) {
```

```

        document.getElementById('addToWishListLinkAjax').href = "";
    } else if (document.getElementById('addToWishListLink')) {
        document.getElementById('addToWishListLink').href = "";
    }

    if (document.getElementById('addToCompareLink')) {
        document.getElementById('addToCompareLink').href = "";
    }
    if(document.getElementById('replaceCartItemAjax')){
        document.getElementById('replaceCartItemAjax').href= "";
    } else if(document.getElementById('replaceCartItemNonAjax')){
        document.getElementById('replaceCartItemNonAjax').href = "";
    }
    categoryDisplayJS.selectedAttributes = new Object();
    categoryDisplayJS.selectedProducts = new Object();
}

```

Example 3-48 illustrates the handling of a delayed hide for the pop-up window. For a delayed hide, we call `delayHidePopup`, which sets the time-out for the pop-up window.

Example 3-48 Setting the proper delay before hiding a pop-up window

```

function delayHidePopup() {

    setTimeout(dojo.hitch(this,"hidePopup",'second_level_category_popup'),200);
}

```

WCDialog (wc.widget.WCDialog)

WcDialog is the extension to dijit.Dialog. It is used to display the mini shopping cart drop-down menu on the header, as shown in Figure 3-15. Specific functions are required to show the mini shopping cart drop-down menu, hide on delay, and show the list of items on mouse over again. To fulfill the requirement, we have extended the dijit.Dialog and created our own WcDialog.

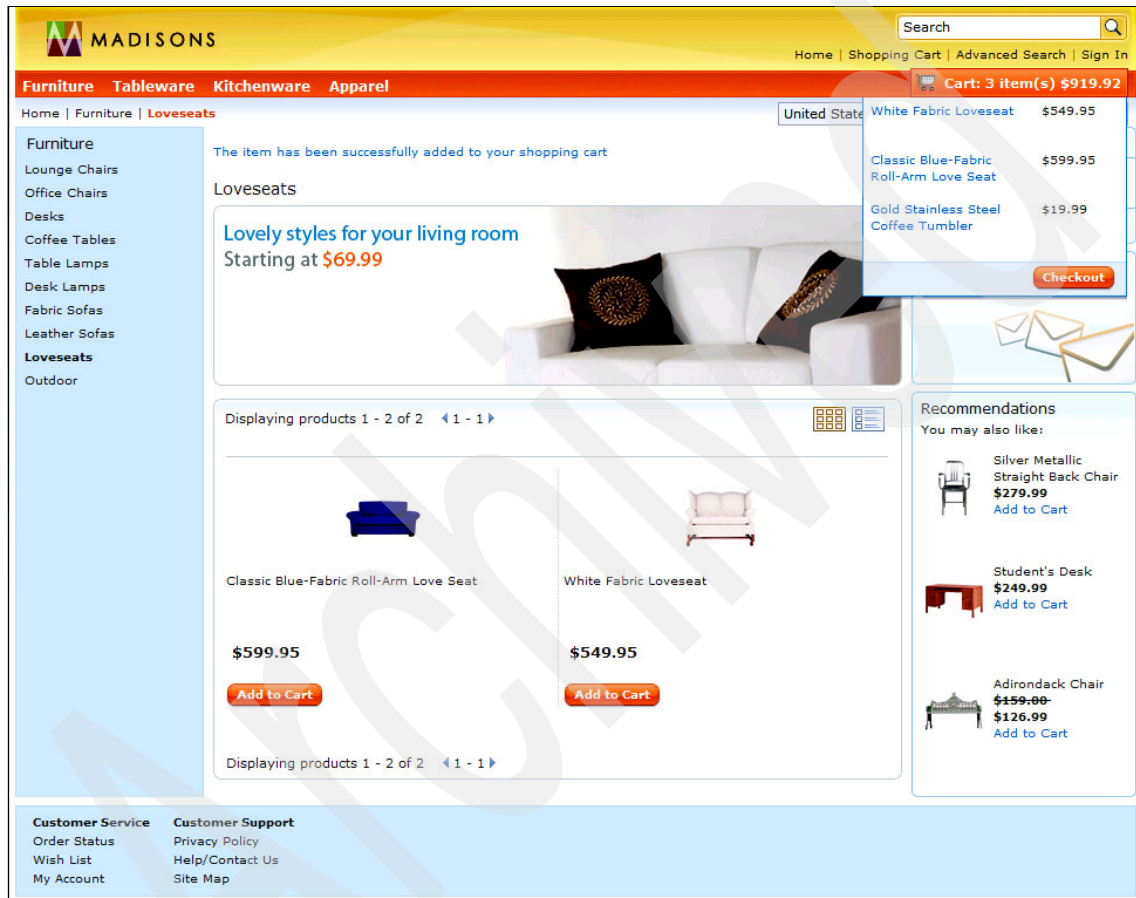


Figure 3-15 MiniShoppingCart drop-down menu

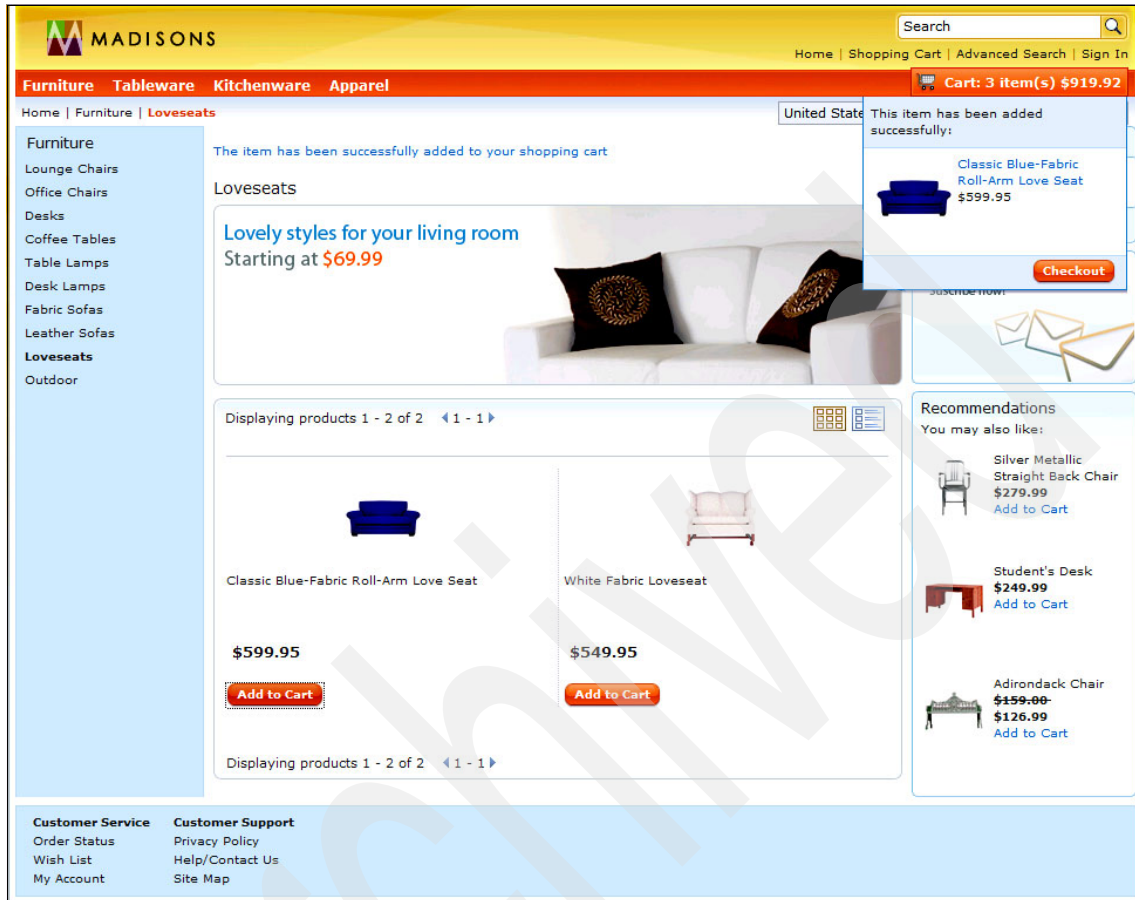


Figure 3-16 ProductAddtoCart drop-down menu

show:function(event)

This method is an override of the Dojo dialog show method.

1. This will first Call Dojo dialog show method.
2. Make entire document to listen to onClick event and allow _onMouseClicked function to handle it.
3. Hide the title bar.
4. Start the timer to close this dialog after timeOut milliseconds if closeOnTimeOut is set to true. If closeOnTimeOut == false, developer has to explicitly call hide(). We never close this dialog.
5. Stop propagating the event further up the document.

_position: function()

This function positions this dialog widget at the given x and y coordinates if passed in.

This function positions this dialog widget relative to the relatedSource element passed in. It can also display the dialog box below the relatedSource element. The left margin is same for both.

If not passed, this will delegate it to parent Dojo dialog to position it at the center of the window.

rePosition: function(source)

The dialog is already created and associated with some other source, reposition it, and reset the timer.

hide: function()

This method will do some initialization operations. If a dialog box is associated with two or more buttons or divs and if it is showing somewhere else, the timer would be set. This method will reset that timer.

getDisplayStatus: function()

We use the displayStatus variable to indicate the current status of the dialog object. If the dialog box is opened on the click of a button, this onclick event will propagate and call the _onMouseClicked function, which closes the dialog box.

We use the displayStatus variable to indicate the current status of the dialog object. If the dialog box is opened on the click of a button, this onclick event will propagate and call the _onMouseClicked function, which closes the dialog box. If we do not stop this event from propagating further up, then the dialog box will not work properly. To avoid this situation we are using displayStatus as a flag and the getDisplayStatus and setDisplayStatus are getter and setter methods for this flag.

cancelCloseOnTimeOut: function()

This variable cancels the timer function that was setup to close the dialog box after timeOut milliseconds.

This function is called from mousePress and onKey event handlers when the event takes places inside the dialog box, so that we do not close the dialog box when the user is using the dialog widget.

_onKey:function(event)

This method checks for the key press event and see if the event is inside dialog. If so user is using the dialog box and so cancel the closeOnTimeOut event. Do not close the dialog box unless user explicitly comes out of it by clicking ESC key or by clicking mouse outside dialog area.

We have to handle it first and then delegate it to Dojo dialog, because Dojo will stop propagating this event and we may not get the chance to handle this if Dojo handles it first.

_onMouseClick: function(*Event*/ evt)

This variable handles mouse click events.

This variable handles mouse click events. If the mouse is clicked inside the dialog widget area, then the cancelling of the closeOnTimeOut feature is executed. This will keep the dialog box from closing when it is scheduled to time out. If mouse is clicked outside dialog widget area then close the dialog immediately.

Sample Usage

Example 3-49 shows how the MiniShopCartDropDown shows the quick details of the current order items. The showMiniShopCartDropDown method will set the coordinates for the dialog box. We then check for the content type to show either a product add drop-down menu (as shown in Figure 3-15 on page 153) or to show quick cart (as shown in Figure 3-16 on page 154) We then call the show method to show the content of the drop-down menu. Internally, this drop-down content is a refresh area, which will get refreshed after each model change (as mentioned in “wc.widget.RefreshArea” on page 106). Example 3-50 on page 157 shows the sample code to destroy the dialog box when the content has been changed, for a particular action, or on delay.

Example 3-49 Showing the mini shopping cart drop-down menu

```
function
showMiniShopCartDropDown(event,relativeId,contentId,contentType){

    //Calculate the X and Y co-ordinates for the dialog. We do not want
it to be at the center of the screen.
    var t = dojo.byId(relativeId);
    var c = dojo.coords(t,true);
    var x1 = c.x;
    var y1 = c.y;

    if(dropDownDlg){
        dropDownDlg.x = x1;
```

```

        dropDownDlg.y = y1;
    }

    /* Dialog is not yet created..Create one */
    if(!dropDownDlg){
        var pane = document.getElementById(contentId);
        dropDownDlg = new wc.widget.WCDialog({relatedSource: relativeId,
x:x1,y:y1},pane);
    }

    if(contentType == 'orderItemsList'){
        dojo.byId("MiniShopCartProductsList").style.display = "block";
        dojo.byId("MiniShopCartProductAdded").style.display = "none";
    }
    else if(contentType == 'orderItemAdded'){
        dojo.byId("MiniShopCartProductsList").style.display = "none";
        dojo.byId("MiniShopCartProductAdded").style.display = "block";
    }
    dropDownDlg.show(event);

    // hides the DialogUnderlayWrapper component, the component that
    // grays out the screen behind,
    // as we do not want the background to be greyed out
    dojo.query('.dijitDialogUnderlayWrapper',
document).forEach(function(tag) {
        tag.style.display='none';
    });
}

```

Example 3-50 Closing the drop-down menu

```

function destroyDialog(){
    //If data has changed, then we should destroy the
    quick_cart_container dialog and recreate it with latest data
    dojo.query('.dijitDialog', document).forEach(function(tag) {
        if (dijit.byNode(tag).id == 'quick_cart_container')
            dijit.byNode(tag).destroyRecursive();// or
    dijit.getEnclosingWidget(tag).destroyRecursive();
    });
    dropDownDlg = null;
}

```

Dropdown(wc.widget.WCDropDownButton)

WcDropDownButton is the extension for dijit.form.DropDownButton. We are using this to show the TopCategories menu in the header, as shown in Figure 3-17. To show this kind of menu, we need functional requirements that cannot be achieved directly through a Dojo widget. We therefore extended the Dojo widget and created our own widget to fulfill the requirement.

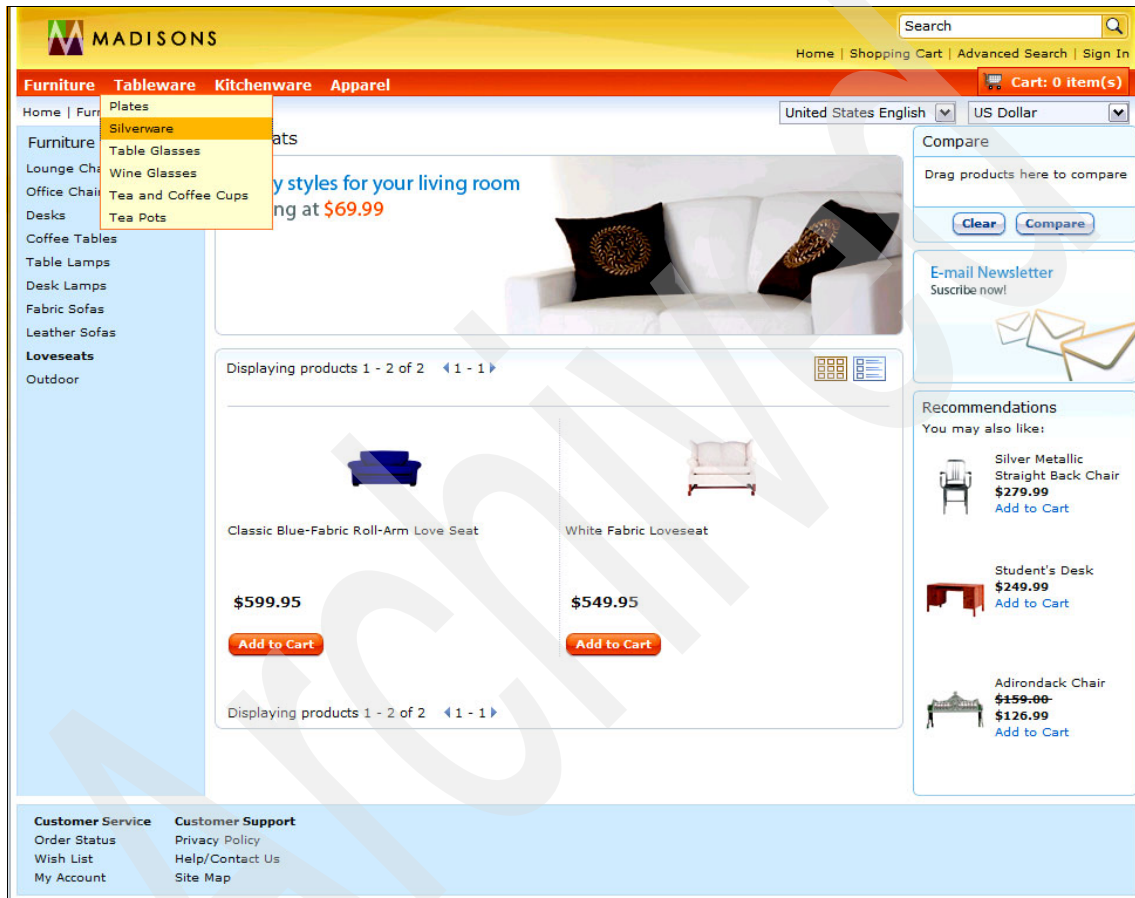


Figure 3-17 TopCategory menu

_onKey: function(*Event*/ e)

This method is called automatically when the user presses a key on the menu pop-up node. This method calls the super method and will check for the key codes and URL. If the current object is an URL, it will locate that URL.

_onDropDownClick: function(*/*Event*/ e)*

This method is called automatically when the user clicks on the menu pop-up node. This method calls the super method and will check for the key codes and URL. If the current object is an URL, it will locate that URL.

Sample Usage

Example 3-51 shows how we are use the WcDropDownButton in combination with the Dojo menu to show the TopCategories menu in the header. For each iteration of the top level loop, we create the wc.widget.WCDropDownButton to show the root icon and a dijit.Menu to hold the menu items for each TopCategory. For each TopCategory, we run another loop, which creates dijit.MenuItem for each SubCategory.

Example 3-51 WCDropDownButton usage in combination with Dojo menu

```
<div id="header_menu_dropdown" style="display:none;">
  <c:forEach var="topCategory" items="${catalog.topCategories}"
varStatus="status">
    <wcf:url var="CategoryDisplayURL" value="Category3">
      <wcf:param name="langId" value="${langId}" />
      <wcf:param name="storeId" value="${WCPParam.storeId}" />
      <wcf:param name="catalogId" value="${WCPParam.catalogId}" />
      <wcf:param name="top" value="Y" />
      <wcf:param name="categoryId" value="${topCategory.categoryId}"
    />
    </wcf:url>

    <fmt:message key="DROPDOWN_ACTIVATE" bundle="${storeText}"
var="categoryTitle">
      <fmt:param value="${topCategory.description.name}" />
    </fmt:message>

    <div dojoType="wc.widget.WCDropDownButton"
url="${CategoryDisplayURL}" title="${categoryTitle}">
      <span><c:out value="${topCategory.description.name}"
escapeXml="false"/></span>
      <div dojoType="dijit.Menu" class="dropdown">
        <c:forEach var="subTopCategory"
items="${topCategory.subCategories}">
          <wcf:url var="subTopCategoryDisplayUrl"
value="Category4">
            <wcf:param name="catalogId"
value="${WCPParam.catalogId}" />
            <wcf:param name="storeId" value="${WCPParam.storeId}"
          />
        />
      </div>
    </div>
  </c:forEach>
</div>
```

```

        <wcf:param name="categoryId"
value="\${subTopCategory.categoryId}" />
        <wcf:param name="langId" value="\${langId}" />
        <wcf:param name="parent_category_rn"
value="\${topCategory.categoryId}" />
        <wcf:param name="top_category"
value="\${topCategory.categoryId}" />
        <wcf:param name="pageView" value="image" />
    </wcf:url>
    <div dojoType="dijit.MenuItem"
onClick="loadLink('${subTopCategoryDisplayUrl}');">
        <span><a href='${subTopCategoryDisplayUrl}'><c:out
value="\${subTopCategory.description.name}"
escapeXml="false"/></a></span>
    </div>
</c:forEach>
</div>
</div>

</c:forEach>
</div>

```

ScrollablePane(wc.widget.ScrollablePane)

The ScrollablePane widget can accept any content, and can scroll this content left and right or up and down. ScrollablePane provides a scrolling effect for the items or image content inside the widget. Each item in the content should be inside a ContentPane widget. The user can scroll the items manually as well as automatically. Passing the autoScroll parameter as true will scroll the items automatically. There are two basic control buttons to allow users to manually scroll the contents left and right as desired. Figure 3-18 on page 161 show how we are using the scrollable widget to show the e-marketing spot on the store homepage. This widget can be used at different places, and for different usages (such as to show product associations and MerchandisingAssociations in product and category pages, and to show the recommendations and special offers in any of the pages). The style and behavior of the widget can be changed by passing a few parameters during the declaration of the widget. This is explained in detail in 3.3.3, “How to customize the Dojo widgets” on page 174.



Figure 3-18 Scrollable widget

identifier: String

This property is the identifier of the widget.

size: Number

This property indicates the size of the widget. This is calculated based on number of nodes to show and width of each item.

autoScroll: Boolean

This property is the flag that indicates whether to scroll automatically or manually.

totalDisplayNodes: Number

This property indicates how many nodes to show at a time.

thumbHeight: Number

This property is the default height of a thumbnail image.

thumbWidth: Number

This property is the default width of a thumbnail image.

isScrollable: Boolean

This property uses smoothScroll to move between pages when set to true.

isHorizontal: Boolean

This property displays thumbnails horizontally if set to true. Otherwise, thumbnails are displayed vertically.

sizeProperty: String

This property indicates the way it will animate.

totalItems: Number

This property indicates the total number of items in the queue.

currentRight: Number

This property indicates the current right item index after a animation.

currentLeft: Number

This property indicates the current left item index after a animation.

leftScroll: Boolean

This property indicates the scrolling direction.

rightScroll:Boolean

This property indicates scrolling direction.

leftMost:Number

This property indicates the left-most item in the queue.

rightMost:Number

This property indicates the right-most item in the queue.

lock:Boolean

This property is the locking variable that locks the animation.

direction:Number

This property is the variable that shows the direction of animation. This can be 1 or -1.

delay:Number

This property indicates the delay of animation.

state:String

This property indicates the state of animation.

altPrev:String

This property indicates alternative text for Previous and Next button images.

templmgPath:String

This property indicates the image path for the widget images.

scrollHorizontal: function(*Object?*/ args)

This property animate horizontal.

description: This method is used for horizontally scrolling

scrollVertical: function(*Object?*/ args)

summary: This is to animate vertical.

This property is used indicate vertical scrolling.

pause: function()

This property pauses the animation. This method is used internally while switching the directions.

play: function()

This property start or restarts the animation. This method is used internally to start or restart the animation after direction changes.

autoScroller: function()

This property is the method providing the auto scroll effect. This method calls prev or next methods based on the direction and will keep auto scrolling.

appendItems: function(items)

This method appends items after a animation. This method is called after a direction change to rearrange the items accordingly.

setDataStore: function(nodeType)

This method gets the nodetype and returns the nodes. This method takes the nodetype as parameter and returns the nodes in the scroll area with that node type.

next: function()

This method scrolls the right-side items. This method sets the direction of the scroll and calls the showThumbs methods to scroll the right-side items.

prev: function()

This method scrolls the left-side items. This method sets the direction of the scroll and calls showThumbs methods to scroll the left-side items.

adjustLeftScroll: function()

This method adjusts left-side nodes after an animation. This method will remove the hidden nodes from the right-side and adds them to left-most position.

adjustRightScroll: function()

This method adjusts right-side nodes after an animation. This method will remove the hidden nodes from the left-side and adds them to right-most position.

showThumbs: function(idx)

This method scrolls the items according to direction. This will animate the items with a specified delay and scale in the given directions, and calls the adjustment methods to get the nodes to adjust for further animations.

loadImage: function(data, position)

This method loads the items from the scrollable area. This method will create new nodes with all data and append them to thumbNode, or insert them first.

updateNavControls: function()

This property updates the navigation controls to hide or show them when at the first or last images.

Example 3-52 shows how to use the ScrollablePane in your page. Each item in the ScrollablePane should be a dijit.layout.ContentPane.

Example 3-52 Using ScrollablePane

```
<div id="id" dojoType="wc.widget.ScrollablePane"
autoScroll='false'
altPrev = '<fmt:message key="SCROLL_LEFT" bundle="${storeText}" />'
altNext = '<fmt:message key="SCROLL_RIGHT" bundle="${storeText}" />'
    <div dojoType="dijit.layout.ContentPane" class="imgContainer">
<HTML markup of the scrollable object>
</div>
<div dojoType="dijit.layout.ContentPane" class="imgContainer">
<HTML markup of the scrollable object>
</div>
</div>
```

RangeSlider(wc.widget.RangeSlider)

This widget defines a two handle slider that can be used to drive range selection for an attribute. This is used to adjust price range selection in the Fast Finder page. Figure 3-19 on page 166 shows the Fast Finder page with the RangeSlider on the left sidebar.

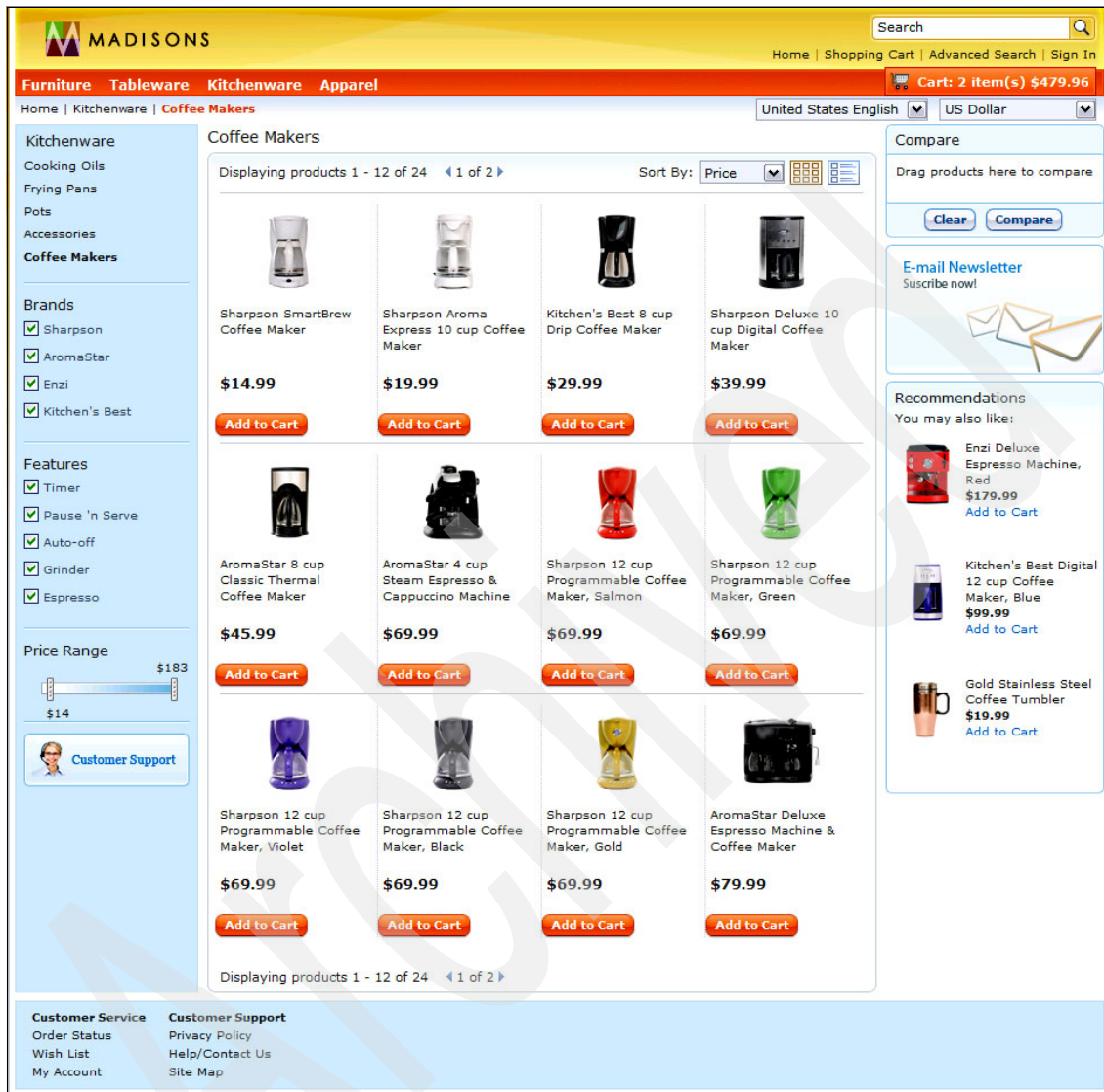


Figure 3-19 Range Slider in the Fast Finder page

defaultStart: Integer

This indicates the default initial position for the first sliding handle.

defaultEnd: Integer

This indicates the default initial position for the second sliding handle.

clickSelect: Boolean

This parameter allows the user to click Bar to move the sliding handles to the nearest point where it was clicked. This parameter defaults to true.

activeDrag: Boolean

This parameter allows the sliding handle will move along with the mouse pointer when user drags it, if it is false then handle will not move by dragging the handle.

incrementValue: Integer

This parameter indicates the multiple that determines values. For example, if it is set to 5, then the values will be in multiples of 5 (0, 5, 10, 15...100).

decimalPoints: Integer

This parameter indicates the number of decimal points to show on the values. Defaults to 0.

showTooltip: Boolean

This parameter defines whether the tooltip is available. Setting this parameter to true indicates the tooltip is available. Setting it to false indicates the tooltip will not be available for display. The default is true.

showTooltipAllTime: Boolean

This parameter defines whether the tooltip is visible. Setting this parameter to true indicates the tooltip is visible. Setting it to false indicates the tooltip is visible only when the user hovers the mouse over the handles. The default is true.

prefix: String

This parameter indicates the prefix for the 'upper' and 'lower' value for the widget.

Suffix: String

This parameter indicates the suffix for the 'upper' and 'lower' value for the widget.

currencyCode: String

This parameter indicates the currency code if the values are prices.

firstHandleTitle: String

This parameter is a translated string that is appended to the left handle and is read by screen readers when the left handle has the focus.

secondHandleTitle: String

This parameter is a translated string that is appended to the right handle and is read by screen readers when the right handle has the focus.

Sample Usage

Example 3-53 shows how we are using the RangeSlider in the Fast Finder page to bring down the items by price range.

Example 3-53 RangeSlider in Fast Finder page

```
<div style="padding-left: 15px;"
id="horizontalRangeSelector"
dojoType="wc.widget.RangeSlider"
  onChangeMade="filterWithPrice"startRange="<c:out
value='${minPrice}' />"totalRange="<c:out value='${maxPrice -
minPrice}' />" prefix="<c:out value='${prefix}' />"
  currencyCode="<c:out value='${CommandContext.currency}' />"
  defaultStart="0"
  defaultEnd="0"
  clickSelect="true"
  snapToGrid="false"
  activeDrag="true"
  incrementValue="1"
  decimalPoints="0"
  showTooltip="true"
  showTooltipAllTime="true"
  firstHandleTitle="<fmt:message key="FF_FIRSTHANDLE_TITLE"
bundle="${storeText}" />"
  secondHandleTitle="<fmt:message key="FF_SECONDHANDLE_TITLE"
bundle="${storeText}" />">
</div>
```

ProgressBar(dijit.ProgressBar)

We are using the ProgressBar in combination with the dialog box in the store front. When the user performs an action triggering a server request, we show a progress bar. This indicates that the process is ongoing. Once the response comes back this is hidden. Figure 3-20 on page 169 shows what the progress bar looks like in the store, and Example 3-54 on page 169 gives you the sample code. Inside the "<div id="progress_bar_popup" dojoType="dijit.Dialog">" you can have your own animated content to show.

We use "cursor_wait" and "cursor_clear" to show and hide the progress bar. These methods can be called from different places in your Javascript code, Example 3-55 on page 170 and Example 3-56 on page 171 shows the full definition of these methods.

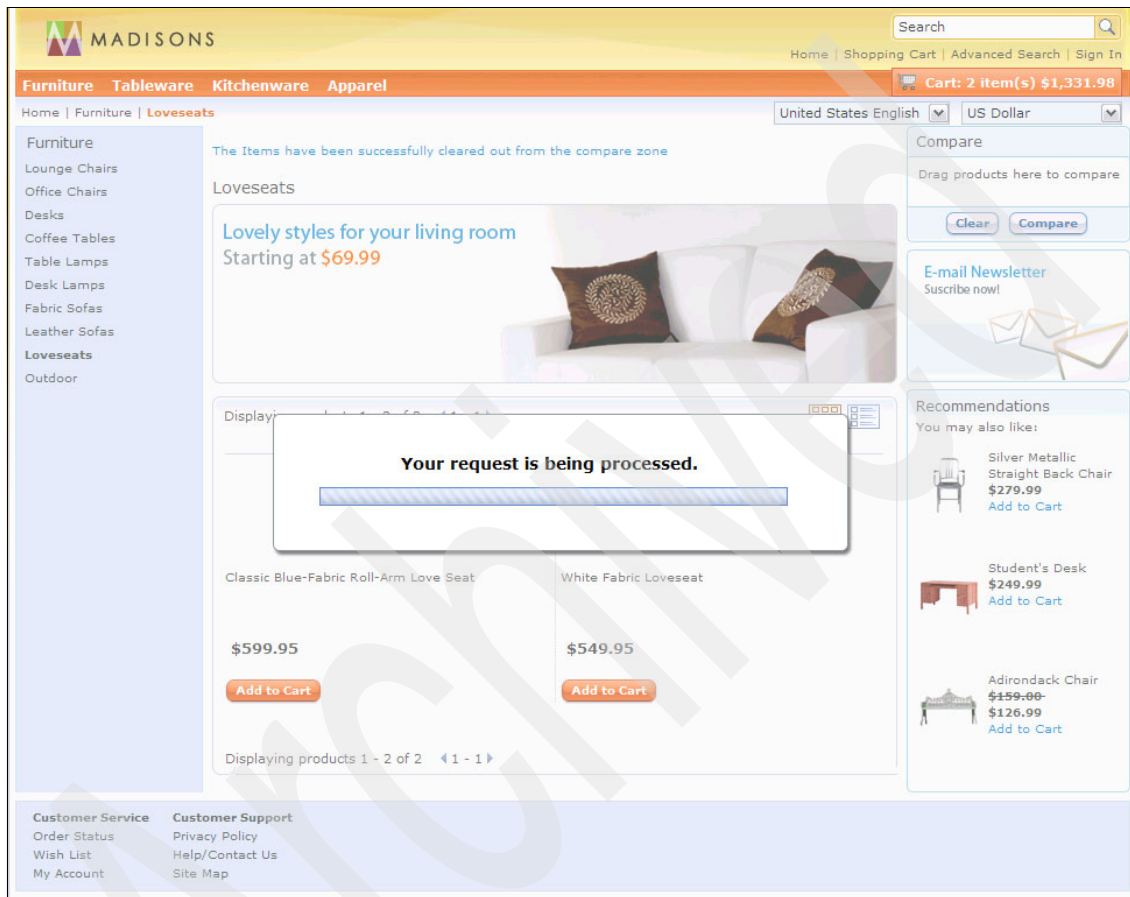


Figure 3-20 Progress bar used in store front

Example 3-54 Using the progress bar in combination with the dialog box

```
<div id="progress_bar_popup" dojoType="dijit.Dialog">
  <div class="top_left">images/trasparent.gif" /></div>
  <div class="top_right">images/trasparent.gif" /></div>
  <div class="header" id="popupHeader"></div>

  <div class="bodyarea">
```

```

        <div class="bodycontent">
        <div class="title">
            <span id="progress_bar_title" tabindex="-1"><fmt:message
key="PROGRESS_BAR_REQUEST_PROCESSED" bundle="${storeText}"/></span>
        </div>

        <div id="progress_bar" style="width:400px" indeterminate="true"
dojoType="dijit.ProgressBar"></div>

        <div class="clear_float"></div>
    </div>
</div>

<div class="footer">
    <div class="bot_left"></div>
    <div class="bot_right"></div>
</div>
</div>

```

Example 3-55 Showing the progress bar

```

function cursor_wait(checkForOpera) {
    busyCount++;
    var showPopup = true;

    //Since dijit does not support Opera
    //Some progress bar dialog will be blocked in Opera to avoid error
    if(checkForOpera == true){
        if(dojito.isOpera > 0){
            showPopup = false;
        }
    }

    //For all other browsers and pages that work with Opera
    //Display the progress bar dialog
    if(showPopup){
        var popup = dijit.byId('progress_bar_popup');

        if(popup != null){
            //Hide the header for the close button
            popup.closeButtonNode.style.display='none';

            //Show the progress bar dialog
            popup.show();
        }
    }
}

```



```

        //For accessibility: set the focus on the progress bar message
        dojo.byId('progress_bar_title').focus();
    }
}
}

```

Example 3-56 This method will hide the progress bar

```

function cursor_clear() {
    busyCount = busyCount - 1;

    if(busyCount < 1){
        //Hide the progress bar dialog
        var popup = dijit.byId('progress_bar_popup');
        if(popup != null){
            popup.hide();
        }
    }
}

```

3.3.2 Implementing your own Dojo widgets

This section explains how to implement your own Dojo widget. Before implementing your own Dojo widget, determine your requirements and try to find a Dojo widget that meets your requirements. Try to customize the widget so that it fulfills your requirements. If you do not find a widget, use the following steps to implement your own Dojo widget.

1. Select your package. Normally, the new Dojo widgets related to WebSphere Commerce is under the WebSphere Commerce package. You can utilize the Dojo widget under the WebSphere Commerce package or you may create your own package.
2. Add your widget template HTML file under the wc/templates folder and add the widget-related images under the wc/images folder.
3. Add your widget .js file under wc/widget.
4. The header section of the widget will contain Dojo.provide, which implies the widget class name you are going to provide, followed by Dojo.require. These statements will import the required support from other Dojo widgets.

The next statement is the class declaration, which contains className, super class info, and the body:

```
dojo.declare("className" , <super class info>, {Body});
```

5. If you are extending the functionality from any specific widget, you can add `dojo.lang.extend(superclass,{body})`. You need to have an `inherits` statement in the header to appear as follows:

```
dojo.inherits("className","superClassName");
```

6. The top part of the body section will have the properties for the widget. You can have a `templateString` property instead of the HTML template file in the step 2 on page 171.
7. (Optional) After the class properties, you can add a constructor for your class.
8. After the constructor, the first method that is called automatically is `postCreate`. From this method, you can do the initializations or can call other methods that will set some basic properties for the widget.
9. Apart from the steps above, you can have your functional methods in the body content.

Example 3-57 to Example 3-60 on page 173 gives you more information about how to implement a widget. In these examples, we are explaining the implementation steps for the Scrollable widget.

Example 3-57 Header section of the widget

```
dojo.provide("wc.widget.ScrollablePane");

dojo.require("dijit._Widget");
dojo.require("dijit._Templated");
dojo.require("dojo.parser");
```

Example 3-58 Widget declaration sample

```
dojo.declare("wc.widget.ScrollablePane",
    [dijit._Widget, dijit._Templated],
    { //body content will be here
});
```

Example 3-59 Extension statements

```
dojo.inherits(wc.widget.ExtendedThumbnailPicker,dojox.image.ThumbnailPi
cker);
dojo.lang.extend()

dojo.lang.extend(dojox.image.ThumbnailPicker, {

    imageStore: null,

    request: null,
```

```

isContainer: true,

thumbHeight: null,

thumbWidth: null,

getClickTopicName: function(arguments){
    // This method can perform some functionlity here and will call
the super methods here like
    // this.inherited("getClickTopicName",arguments);
    // or it can use
this.superclass.getclickTopicName.call(arguments);
},

    // and other extended and new methods will follow here

});

```

Example 3-60 Properties declaration for the widget

```

identifier:null,
    //size:Number
    // Size of the widget , this will be calculated based on number of
nodes to show and wid of each item
    size: 0,
    //autoScroll: Boolean
    //This is the flag which indicates whether to autoscroll or manual
autoScroll: false,
    //delay:Number
    //this the delay of animation
delay:2000,
    //state:String
    //this indicates the state of animation
state: null,
    //altPrev:String
    //alternative text for prev and next button images
altPrev: 'Scroll Left',
altNext: 'Scroll Right',
    //templateString:String
    //this is the template string for the widget
templateString:"<div dojoAttachPoint=\"outerNode\"
class=\"thumbOuter\">\n\t<div dojoAttachPoint=\"navPrev\"
tabindex=\"0\" class=\"navPrev\"><img src=\"\"
dojoAttachPoint=\"navPrevImg\" alt=\"\"/></div>\n\t<div

```

```

dojoAttachPoint="thumbScroller\" class="thumbScroller\"
valign="middle\" >\n\t <div dojoAttachPoint="thumbsNode\"
class="thumbsNode\"></div>\n\t</div>\n\t<div
dojoAttachPoint="navNext\" tabindex="0\" class="navNext\"><img
src=""\" dojoAttachPoint="navNextImg\" alt=""\"/></div>\n</div>\n",
    //tempImagePath:String
    //image path for the widget images
    tempImagePath: dojo.moduleUrl("wc.widget", "images/trasparent.gif"),

```

Note: The complete code for the Scrollable widget (ScrollablePane.js) is available in Appendix A, “Additional material” on page 237.

3.3.3 How to customize the Dojo widgets

The widgets mentioned in 3.3.1, “The Dojo widgets used for Web 2.0 Store” on page 135 are highly customizable and flexible. This section will explain how to customize widgets to fit your requirements. You can reuse the widget at different place in a different manner without modifying the actual code of the widget. We have parameterized the widget properties that can be passed externally. The styles were taken from the CSS class that can be set in your CSS files, so you can change the look and feel of the widget without modifying the actual code. And if you want to add additional functionality to the widget, you can extend the widget (as shown in Example 3-59 on page 172).

A regular scrollable widget with a horizontal scrolling effect will look similar to Figure 3-21 on page 175. To show the scrollable widget with horizontal scrolling effect, have CSS classes defined as shown in Example 3-61 on page 175. The JSP code will look similar to Example 3-62 on page 177.



Figure 3-21 Regular scrollable pane used for homepage e-spot

Example 3-61 Scrollable widget with horizontal scrolling effect: CSS classes

```
#scroll_ad {
    min-width:585px;
```

```

padding-right:0px;
margin:0px;
vertical-align:top;
height: 200px;
}
.thumbOuter{
border:0;
height: 200px;
width: 500px;
}
.thumbOuter div{
border: 0px;
}
.navPrev {
float:left;
cursor: pointer;
width: 20px;
height: 100%;
background:
url("../images/colors/color1/accessories_images_arrow_left.png")
no-repeat center center;
padding: 0px 0px 10px 0px;
}
.navNext {
float:right;
cursor: pointer;
width: 20px;
height: 100%;
background:
url("../images/colors/color1/accessories_images_arrow_right.png")
no-repeat center center;
padding: 0px 0px 10px 0px;
}
.thumbScroller {
border: 0px;
float:left;
overflow:hidden;
position: relative;
height: 100%;
padding: 0px 10px 0px 7px;
}
.thumbsNode{
border: 0px;
height: 100%;
float:left;

```

```

}
.imgContainer {
    float:left;
    border: 0px;
    width: 135px;
    height: 100%;
}

```

Example 3-62 Scrollable widget with horizontal scrolling effect: JSP code

```

<div id="id" dojoType="wc.widget.ScrollablePane" autoScroll='false'
altPrev = '<fmt:message key="SCROLL_LEFT" bundle="${storeText}" />'
altNext = '<fmt:message key="SCROLL_RIGHT" bundle="${storeText}" />'>
    <div dojoType="dijit.layout.ContentPane" class="imgContainer">
        <!-- content goes here -->
    </div>
    <div dojoType="dijit.layout.ContentPane" class="imgContainer">
        <!-- content goes here -->
    </div>
    <div dojoType="dijit.layout.ContentPane" class="imgContainer">
        <!-- content goes here -->
    </div>
    <div dojoType="dijit.layout.ContentPane" class="imgContainer">
        <!-- content goes here -->
    </div>
    <div dojoType="dijit.layout.ContentPane" class="imgContainer">
        <!-- content goes here -->
    </div>
</div>

```

The scrollable widget will fit in to other requirements of the store by modifying the CSS class. In the product display page, to show the Merchandising Associations as a single item at a time (as shown in Figure 3-22 on page 178), modify the changes shown in example Example 3-63 on page 178 and Example 3-64 on page 180.



Figure 3-22 Customized Scrollable to show one item in the product page

Example 3-63 CSS changes to single item scrolling

```
.thumbOuter{
    border:0;
    height: 150px;
    width: 200px;
}
.thumbOuter div{
    border: 0px;
}
```



```

.navPrev {
    float:left;
    cursor: pointer;
    width: 20px;
    height: 100%;
    background:
url("../images/colors/color1/accessories_images_arrow_left.png") no-repeat
center center;
    padding: 0px 0px 10px 0px;
}

.navNext {
    float:right;
    cursor: pointer;
    width: 20px;
    height: 100%;
    background:
url("../images/colors/color1/accessories_images_arrow_right.png")
no-repeat center center;
    padding: 0px 0px 10px 0px;
}

.thumbScroller {
    border: 0px;
    float:left;
    overflow:hidden;
    position: relative;
    height: 100%;
    padding: 0px 10px 0px 7px;
}

.thumbsNode{
    border: 0px;
    height: 100%;
    float:left;
}

.imgContainer {
    float:left;
    border: 0px;
    width: 200px;
    height: 100%;
}

```

Example 3-64 Scrollable to show one item at a time

```
<div id="id" dojoType="wc.widget.ScrollablePane" autoScroll='true'
totalDisplayNodes="1" thumbHeight = "150" thumbWidth="150" sizeProperty
= "width" delay = "500" altPrev = '<fmt:message key="SCROLL_LEFT"
bundle="${storeText}" />' altNext = '<fmt:message key="SCROLL_RIGHT"
bundle="${storeText}" />'>
    <%out.flush();%>
    <c:import
url="${jspStoreDir}Snippets/Marketing/ESpot/ScrollingProductsESpot.jsp"
>
        <c:param name="emsName" value="HomePageFeaturedProducts" />
        <c:param name="scrollable" value="true" />
        <c:param name="catalogId" value="${WCParam.catalogId}" />
    </c:import>
    <%out.flush();%>
</div>
```

If page needs to show a vertical scroll instead of horizontal, as shown in Figure 3-23, the CSS and JSP code should look similar to Example 3-65 and Example 3-66 on page 183.

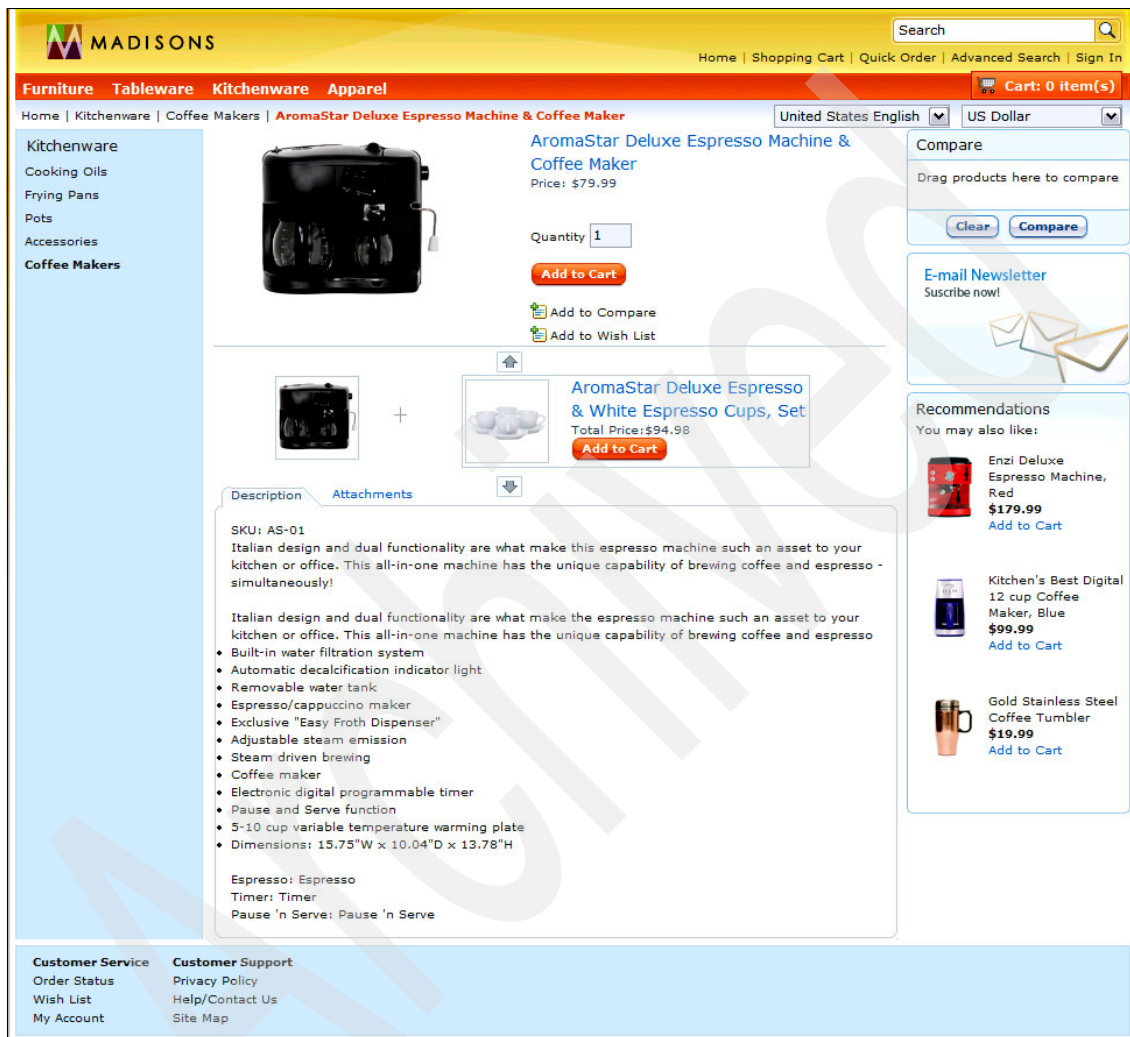


Figure 3-23 Scrollable customized to scroll vertical

Example 3-65 CSS change for vertical scrolling

```
.thumbOuter{
    border:0;
    height: 150px;
    width: 200px;
```

```

}
.thumbOuter div{
    border: 0px;
}
.navPrev {
    vertical-align:top;
    cursor: pointer;
    width: 20px;
    background:
url("../images/colors/color1/accessories_images_arrow_up.png")
no-repeat center center;
    padding: 10px 0px 0px 0px;
}

.navNext {
    vertical-align:bottom;
    cursor: pointer;
    width: 20px;
    background:
url("../images/colors/color1/accessories_images_arrow_down.png")
no-repeat center center;
    padding: 10px 0px 0px 0px;
}

.thumbScroller {
    border: 0px;
    overflow:hidden;
    position: relative;
    height: 100%;
}

.thumbsNode{
    border: 0px;
    height: 100%;
}

.imgContainer {
    border: 0px;
    width: 200px;
    height: 100%;
}

```

```
<div id="id" dojoType="wc.widget.ScrollablePane" autoScroll='false'
isHorizontal ="false" totalDisplayNodes="1" thumbHeight = "150"
thumbWidth="150" sizeProperty = "height" altPrev = '<fmt:message
key="SCROLL_UP" bundle="${storeText}" />' altNext = '<fmt:message
key="SCROLL_DOWN" bundle="${storeText}" />'>
    <%out.flush();%>
    <c:import
url="${jspStoreDir}Snippets/Marketing/ESpot/ScrollingProductsESpot.jsp"
>
        <c:param name="emsName" value="HomePageFeaturedProducts" />
        <c:param name="scrollable" value="true" />
        <c:param name="catalogId" value="${WCParam.catalogId}" />
    </c:import>
    <%out.flush();%>
</div>
```

3.4 Introduce Web 2.0 features into your Web 1.0 Store

This section provides instructions on how to introduce Web 2.0 elements into your Web 1.0 Store. To have Web 2.0 capabilities, follow the prerequisites described in 3.4.1, “Adding a Web 2.0 feature in your Web 1.0 Store” on page 183. This section also reviews the relationships between the Dojo widgets and the interactions that occur in this scenario. We add the ability to drag-and-drop products into the mini-shopping cart to the ConsumerDirect store, and have the mini shopping cart dynamically update with the new shopping cart information.

3.4.1 Adding a Web 2.0 feature in your Web 1.0 Store

In this section, the goal is to become more familiar with the Web 2.0 Ajax programming framework and Dojo widgets, and provide instructions to introduce Web 2.0 elements into your Web 1.0 Store.

We specifically add the ability to drag-and-drop products into the mini-shopping cart to the ConsumerDirect store, and have the mini shopping cart dynamically update with the new shopping cart information.

This section also reviews the relationships between the Dojo widgets and the interactions that occur in this scenario.

Determining the sections in the page that need to be updated

In this part, we identify the pages or fragments that need to be updated, as shown in Figure 3-24. The goal is as follows:

1. Create a drop zone on top of the mini shopping cart.
2. Allow items to be dragged to the mini shopping cart.
3. Execute an Add to Cart-type command upon an item drop.
4. Refresh the mini shopping cart fragment once the item has been dropped.

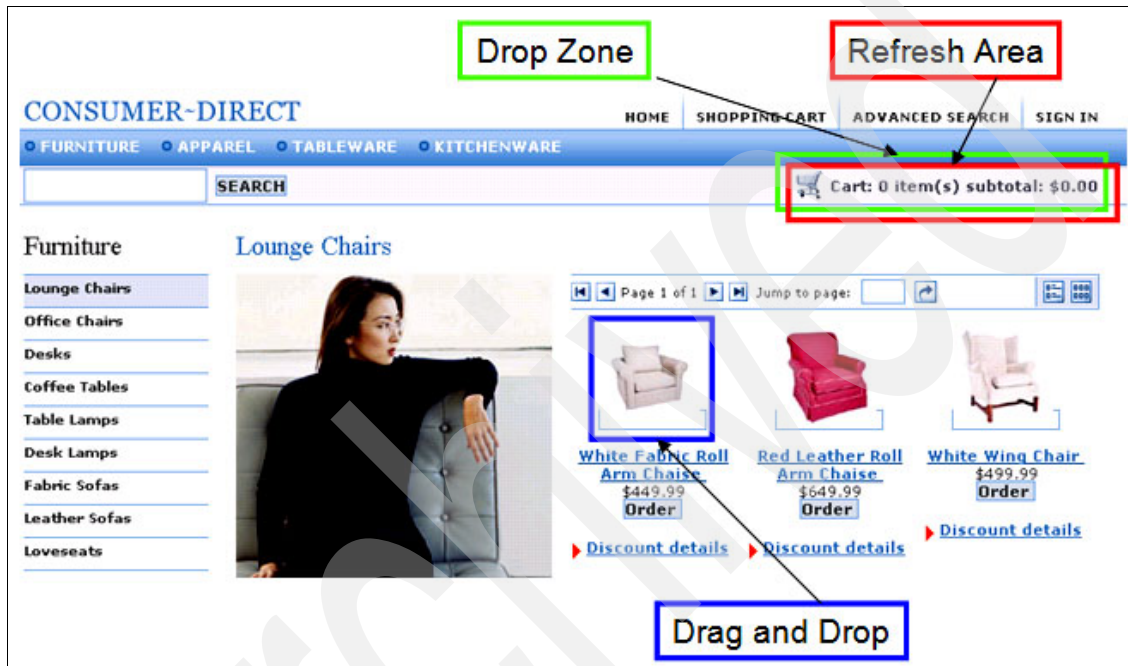


Figure 3-24 Drag and drop parts in consumer direct store

Let us look at the page and see what we want to accomplish.

Let us identify the code changes, and the interactions involved

1. We have made the catalog images as drag sources which will have the drag type associated with them.
2. A drop zone was created to receive the product images.
3. A function was written, which will execute on a drop.
4. Subscribe the javascript which does the client side process and invokes the service with drop event.

5. An Ajax Dojo service is defined, which is executed by the function and will request a URL against the Commerce server. When completed successfully, a Dojo event is created.
6. AjaxOrderChangeServiceItemAdd URL is defined in our Struts configuration.
7. Include all dojo require code in the jsp.
8. A Refresh Area was created around the mini shopping cart section of the page, which is linked to the refresh controller.
9. A Render context was created and linked to the Refresh Controller widget on the page.
10. A Refresh Controller was created, which will decide when it is appropriate to execute a URL/view to be rendered into the refresh area.
11. A new view was created in our Struts configuration that corresponds to the URL/view that will display the updated shopping cart.
12. Access control was implemented, which is required for access to our new view.

Make the required code changes

1. In this section, we will access the drag and drop source to change the thumbnail image into a draggable item. You must add the dojo.dnd.Source tag on top of the image tag as shown in Example 3-67. We pass the parameters copyOnly, which indicates that the image cannot be moved it can just be copied. dndType which sets the type of the source, is checked during the process once its dropped to the target.

Example 3-67

```
<div dojoType="dojo.dnd.Source" jsId="dndSource"
id="{catEntryIdentifier}" copyOnly="true" dndType="{dragType}">
<div class="dojoDndItem" dndType="{dragType}"
id="WC_CatalogEntryDBThumbnailDisplayf_div_2">
<c:choose>
<c:when test="{!empty catalogEntry.description.thumbNail}">
<a href="<c:out value="{catEntryDisplayUrl}" escapeXml="false"/>"
id="img<c:out value="{catEntryIdentifier}"/>" class="itemhover"
onfocus="showPopupButton('{prefix}_{catEntryIdentifier}');">
"
alt="<c:out value="{fn:replace(catalogEntry.description.name,
search, replaceStr)} {displayPriceString}" escapeXml="false"/>"
border="0"/>
/a>
```

```

</c:when>
<c:otherwise>
<a href="<c:out value="\${catEntryDisplayUrl}" escapeXml="false"/>"
id="img<c:out value="\${catEntryIdentifier}"/>" class="itemhover"
onfocus="showPopupButton('${prefix}_\${catEntryIdentifier}');">
images/NoImageIcon_sm.jpg"
alt="<c:out value="\${fn:replace(catalogEntry.description.name,
search, replaceStr)} \${displayPriceString}" escapeXml="false"/>"
border="0"/>
</a>
</c:otherwise>
</c:choose>
</div>
</div>

```

2. In this section, we focus on the drop zone within a Web page. A drop zone uniquely identify an HTML DOM element to be the desired drop zone. In Figure 3-24 on page 184, the drop zone is around the mini shop cart. Thus, we can define the drop in the `CachedHeaderDisplay.jsp`, which is the parent JSP that includes the `MiniShopCartDisplay.jsp`. you can define the drop zone area by updating
`<Stores>\ConsumerDirect\include\styles\style1\CachedHeaderDisplay.jsp`
with code shown in Example 3-68.

Example 3-68

```

<div dojoType="dojo.dnd.Target" jsId="miniShopCart_dndTarget"
id="miniShopCart_dndTarget" accept="item, product, package, bundle"
>
<%out.flush();%>
<c:import url="\${jspStoreDir}include/MiniShopCartDisplay.jsp"/>
<%out.flush();%>
</div>

```

This statement will define the drop target with the accept type as item, product, package and bundle. This means that all of these types are acceptable and the user can drag and drop these types into the mini shop cart.

3. Define the `AddItem2ShopCartAjax` method, which follow the process after item drops in the target. This is where we invoke the actual service request. A typical `AddItem2ShopCartAjax` method looks like Example 3-69.

Example 3-69

```

AddItem2ShopCartAjax : function(catEntryIdentifier , quantity)

```



```

{
// summary : This function is used to add an item to cart
using ajax call.
// description : This function is used to add an item to
cart using ajax call.
// catalogEntryId: catentryid of the item.
// quantity: quantity of the item.
var params = [];
params.storeId= this.storeId;
params.catalogId= this.catalogId;
params.langId= this.langId;
params.orderId= ".";
params.catEntryId= catEntryIdentifier;
params.quantity= quantity;
var invalidQTY = new RegExp(/^\d*$/);
if(params.quantity == 0 || params.quantity == "" ||
!invalidQTY.test(params.quantity)){
ErrorHandler.displayErrorMessage(ErrorHandler.errorMessages['QUANTIT
Y_INPUT_ERROR']); return;}
if(this.ajaxShopCart){
wc.service.invoke("AjaxAddOrderItem", params);
this.baseItemAddedToCart=true;
}else{
wc.service.invoke("AjaxAddOrderItem_shopCart", params);
this.baseItemAddedToCart=true;
}
}
cursor_wait()

```

Note: We could have easily just called the regular OrderItemAdd through Ajax rather than this new AjaxOrderChangeServiceItemAdd URL which calls the new WebSphere Commerce order service. The wc.declare.service Ajax API can be used to call any WebSphere Commerce command or service as long as they are defined in the Struts configuration files as
type=com.ibm.commerce.struts.AjaxAction or
type=com.ibm.commerce.struts.AjaxComponentServiceAction.

4. Once the item is dropped into the drop zone, call the actual functional methods that perform server calls. Subscribe these methods with drop operation, as shown in Example 3-70, so that these methods follow just after the item dropped.

Example 3-70

```

function initShopcartTarget(){
dojo.subscribe("/dnd/drop", function(source, nodes, copy,

```

```

target){
target.deleteSelectedNodes();
if(target.parent.id=='miniShopCart_dndTarget'){
if(source.node.getAttribute('dndType')== 'item' ||
source.node.getAttribute('dndType')== 'package') {
AddItem2ShopCartAjax(source.parent.id ,1);
} else if(source.node.getAttribute('dndType')== 'product'
|| source.node.getAttribute('dndType')== 'bundle') {
showPopup(source.parent.id, '${WCPParam.storeId}', '${WCPParam.langId}',
'${WCPParam.catalogId}', function(e){return
e;}, 'miniShopCart_dndTarget');
}
}
});
};
dojo.addOnLoad(initShopcartTarget);

```

We are subscribed the set of statements to DnD. Once the item is dropped the subscribed method will get called. Finally, we have defined a function `AddItem2ShopCartAjax` which makes the actual operations for add to cart , and in this method, we invoke the `AjaxAddOrderItem` service to add a quantity of one of this product to the current pending order.

5. As mentioned in 3.2, “WebSphere Commerce Ajax framework for Dojo” on page 105, the `wc.service.declare` will give you a service object which has a reference to the server action and the success and failure handlers. Example 3-71 shows a typical service declaration for the Add to Cart action.

Example 3-71

```

wc.service.declare({
id: "AjaxAddOrderItem",
actionId: "AjaxAddOrderItem",
url: "AjaxOrderChangeServiceItemAdd",
formId: ""
,successHandler: function(serviceResponse) {
ErrorHandler.hideAndClearMessage();
ErrorHandler.displayStatusMessage(ErrorHandler.errorMessages["SHOPCA
RT_ADDED"]);
}
,failureHandler: function(serviceResponse) {
if (serviceResponse.errorMessage) {
ErrorHandler.displayErrorMessage(serviceResponse.errorMessage);
}
else {
if (serviceResponse.errorMessageKey) {

```

```

ErrorHandler.displayErrorMessage(serviceResponse.errorMessageKey);
}
}
cursor_clear();
}
}),

```

We have created an AjaxAddOrderItem Ajax service object. This service called the AjaxOrderChangeServiceItemAdd URL using Ajax (rather than a normal browser HTTP request) with the parameters in the wc.service.invoke call of our drop event handler. After a successful execution of the Ajax service, this widget will automatically trigger a modelChanged events handler in the RefreshController. See 3.2.1, “Getting familiar with the Ajax Framework extension” on page 106 to understanding how events are triggered and how to register listeners.

6. Ensure that the AjaxOrderChangeServiceItemAdd URL is defined in our Struts configuration as shown in Example 3-72.

Example 3-72

```

<Stores>\WEB-INF\struts-config-order-services.xml
<action parameter="order.addOrderItem"
path="/AjaxOrderChangeServiceItemAdd"
type="com.ibm.commerce.struts.AjaxComponentServiceAction">
<set-property property="authenticate" value="0:0"/>
<set-property property="https" value="0:1"/>
</action>

```

7. Import or define the Dojo code, so you do not have any problems loading the widgets already defined in the example (as well as the ones used in the next steps). Include the statements in the Example 3-73 after the <!--START HEADER--> line of the CachedHeaderDisplay.jsp.

Example 3-73

```

<script type="text/javascript" src="/wcsstore/dojo101/dojo.js
djConfig="true"></script>
<script type="text/javascript">
dojo.registerModulePath("wc", "../wc");
dojo.require("wc.service.common");
dojo.require("dojo.parser");
dojo.require("dojo.dnd.Source");
dojo.require("wc.widget.RefreshArea");
dojo.require("wc.render.RefreshController");
dojo.require("wc.render.Context");

```

</script>

8. Define the elements that allow the mini shopping cart to refresh. As mentioned in 3.2, “WebSphere Commerce Ajax framework for Dojo” on page 105, the Refresh Area widget is a section of the page that will get updated with new content coming from a URL. Within the miniDropZone we created earlier in the CachedHeaderDisplay.jsp, create a refresh area using the example code in Example 3-74. Create the Refresh area which can be reloaded on a successful drop. This refresh area should be added inside the target to keep the Result area always a drop target.

Example 3-74

```
<div dojoType="dojo.dnd.Target" jsId="miniShopCart_dndTarget"
id="miniShopCart_dndTarget" accept="item, product, package, bundle"
>
<div dojoType="wc.widget.RefreshArea" id="MiniShoppingCart"
widgetId="MiniShoppingCart"
controllerId="MiniShoppingCartController" role="wairole:region"
waistate:live="polite" waistate:atomic="true"
waistate:relevant="all">
<%out.flush();%>
<c:import url="{jspmStoreDir}include/MiniShopCartDisplay.jsp"/>
<%out.flush();%>
</div>
</div>
```

9. Define the Refresh Controller and Render. The Refresh area is linked to a Refresh controller, which needs to be related to a render context. The Render Context is simply a mandatory element that helps keeping track of a client state, and is not used in this particular scenario. Add the following to the end of the CachedHeaderDisplay.jsp to define the Render Context, we again use the JavaScript to declare these objects as shown in Example 3-75.

Example 3-75

```
wc.render.declareContext("MiniShoppingCartContext",null,"");
```

10. Define the Refresh Controller. Add the URL declaration as shown in Example 3-76 code to the CacheHeaderDisplay.jsp after the definition of the Render Context (which was added in the previous step).

Example 3-76

```
<c:url var="miniShopcartURL" value="MiniShoppingCartView">
<c:param name="langId" value="{langId}"/>
<c:param name="storeId" value="{storeId}"/>
```

```
<c:param name="catalogId" value="{catalogId}"/>
</c:url>
```

11. Declare refresh controller, as shown in Example 3-77, that takes care of deciding when to refresh the mini shopping cart area.

Example 3-77

```
wc.render.declareRefreshController({
  id: "MiniShoppingCartController",
  renderContext:
wc.render.getContextById("MiniShoppingCartContext"),
  url: "${miniShopcartURL}",
  formId: ""
  ,modelChangedHandler: function(message, widget) {
    var controller = this;
    var renderContext = this.renderContext;
    if(message.actionId in order_updated){
      var param = [];
      if(message.actionId == 'AjaxAddOrderItem'){
        param.addedOrderItemId = message.orderItemId + "";
        showDropdown = true;
      }
      widget.refresh(param);
    }
    else{
      //alert(message.actionId +" not in order_updated..no
      work...");
    }
  }
  ,renderContextChangedHandler: function(message, widget) {
    var controller = this;
    var renderContext = this.renderContext;
  }
  ,postRefreshHandler: function(widget) {
    var controller = this;
    var renderContext = this.renderContext;
    //The dialog contents has changed..so destroy the old dialog
    with stale data..
    destroyDialog();
    if(showDropdown){
      //We have added item to cart..So display the drop down with
      item added message..
      showMiniShopCartDropDown(null,"placeholder",'quick_cart_container','
      orderItemAdded');
```

```

showDropdown = false;
}
cursor_clear();
}
}),

```

In the example above, we have defined the Refresh Controller linked to our Render Context. The id of the Refresh Controller is the controller id that the Refresh Area is linked. The Web 2.0 framework will automatically execute the `modelChangedScript` of the refresh controllers whenever a successful Ajax service is completed. The framework has automatically created listeners in the Refresh Controller for the `modelChanged` event. In this refresh controller widget `modelChangedScript` method, we are going to make sure that the successful Ajax service is actually the `AjaxAddOrderItem` before we ask the Refresh Area widget to get its new contents. The `widget.refresh()` call will cause our Refresh Area to refresh using the `MiniShoppingCartView`, which is a JSP that returns the updated HTML to be inserted into the Refresh Area. The refresh area contents are retrieved through Ajax.

12. Define the `MiniShoppingCartView`, which is executed when the refresh is executed. We need to define this new view in our Struts configuration as shown in Example 3-78. It will simply re-execute the `MiniShopCartDisplay.jsp` file. Add the following definitions to the appropriate areas of the `<Stores>\WEB-INF\struts-config-ext.xml` file.

Example 3-78

In the `<global-forwards>` element add:

```

<forward className="com.ibm.commerce.struts.ECActionForward"
name="MiniShoppingCartView/10551"
path="/include/MiniShopCartDisplay.jsp"/>

```

In the `<action-mappings>` element add:

```

<action path="/MiniShoppingCartView"
type="com.ibm.commerce.struts.BaseAction"/>

```

In the `<global-forwards>` element add:

```

<forward className="com.ibm.commerce.struts.ECActionForward"
name="MiniShoppingCartView/10551"
path="/include/MiniShopCartDisplay.jsp"/>

```

In the `<action-mappings>` element add:

```

<action path="/MiniShoppingCartView"

```

```
type="com.ibm.commerce.struts.BaseAction"/>
```

13. Ensure that you have loaded the appropriate access control policy related to the new view we have added to refresh the MiniShopCart. Add the content from Example 3-79 in to a file web20policy.xml. Use the acpload utility to load the following policy:

Example 3-79

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<!DOCTYPE Policies SYSTEM "../dtd/accesscontrolpolicies.dtd">
<Policies>
<Action Name="MiniShoppingCartView"
CommandName="MiniShoppingCartView">
</Action>
<ActionGroup Name="ConsumerDirectAllUsersViews"
OwnerID="RootOrganization">
<ActionGroupAction Name="MiniShoppingCartView"/>
</ActionGroup>
</Policies>
```

14. Copy the web20policy.xml file to
C:\WebSphere\CommerceServer60\xml\policies\xml
15. Run acpload from the C:\WebSphere\CommerceServer60\bin directory as follows: acpload.bat mall build diet4coke web20policy.xml

Validating the changes

Once the code is changed and saved, ensure all changes are recompiled by the system, and that the Struts and access control changes are picked up.

1. Stop the WebSphere Commerce server if it is running.
2. Clear the <temp>\ConsumerDirect directory if it is populated, to ensure the changes are picked up.
3. Start the WebSphere Commerce server.
4. Navigate to the Furniture; Lounge Chairs category.
5. Drag an item into the shopping cart.

The mini shopping cart fragment at the top of the page should update to reflect the new item added to the shopping cart. You have added Web 2.0 functionality to your Web 1.0 Store.



Testing and debugging your Web 2.0 Store

This chapter describes best practices for testing and debugging the functionalities of your Web 2.0 Store, and presents Web 2.0 Store optimization best practices.

4.1 Testing your Web 2.0 Store

After you develop your store, test the functions and responses of the Web 2.0 Store to verify everything works as expected. Based on the use cases, test all possible flows, including the exception flows, to ensure that the store is functionally tested. Additionally, the store administrator and store management should put the store administration tool to a test. From a performance perspective, stress test with multiple users concurrently accessing the store and cases with large objects (for example, a single user placing an order with large quantity). Other tests to consider are as follows:

- ▶ Accessibility test
- ▶ Globalization test
- ▶ Usability test

In this section, we discuss common Web 2.0 Store tests.

4.1.1 Functional testing

Functional testing is the base of the testing life cycle. Only after you are sure the Web 2.0 Store is functional can you consider the other tests. This section presents some typical functional scenarios of the WebSphere Commerce Web 2.0 Store solution for your reference during functional testing.

Functional scenarios

The following list presents common functional scenarios to be tested.

- ▶ Register a user

There are two places where the registration functionality is provided:

- Store home page
- Order confirmation page

The customer can perform the registration operation when on the home page, or on the order confirmation page after placing an order as a guest user.

To complete the registration, the customer needs to enter relevant information and click a button for submission. If the customer enters an existing user ID an error message displays, indicating this error. For some client-side validation errors, for example, when the customer inputs an invalid character for the age information, the error message should display beside the problem field.

► Manage account information

If the customer logs onto the store as a registered user, he is able to manage his account information. This includes, for example, updating personal information, updating the address book, removing an address, viewing order history, creating new addresses, and so forth.

► Add item to shopping cart

There are two facets to this scenario:

- Add an item by clicking the **Add to Cart** button

The shopper clicks **Add to Cart** from the department page, category page, product compare page, Fast Finder page, e-marketing spot, product detail page, or tooltip of the product quickview. The item should be added to the shopping cart successfully. At the same time, the mini shopping cart details header on the top right corner of the page should be updated to show the current number of items in the cart and the total price. At this time, an order can be created successfully.

- Add an item by dragging an item to the mini shopping cart

The shopper selects an item and drags it to the mini shopping cart area on the top right corner of the page. The item is added to the shopping cart. The mini shopping cart details header on the top right corner of the page should be updated to show the current number of items in the cart and the total price. .

► View items in the shopping cart

A drop-down list containing items in the shopping cart, along with the price of each item, is displayed. The drop-down list also contains a button for check-out at the end. The product details page is displayed when the shopper clicks on a specific item in the shopping cart.

► Update items in the shopping cart

A shopper can perform the following tasks in the shopping cart page:

- Add

The item is added to shopping cart and the shopping cart page is refreshed to show the current number of items and total price.

- Delete

The item is removed from the shopping cart and the shopping cart page is refreshed to show the current number of items and total price.

- Update

The shopping cart is updated and the shopping cart page is refreshed to show the current number of items and total price.

► Choose shipping and payment methods

There are four facets to this scenario:

- Ship items to one address and single payment method
 - **Scenario:** In the testing, you can put three or more items in the shopping cart.
 - **Process:** The process is as follows:
 1. Click the **Single Shipment** button in the shopping cart.
 2. Select one shipping address and shipping method for all items.
 3. Choose single payment method.
 - **Results:** The item details area is refreshed and all the items have the same address and shipping method. The order is created successfully.
- Ship items to one address and multiple payment methods
 - **Scenario:** In the testing, you can put three or more items in the shopping cart.
 - **Process:** The process is as follows:
 1. Click the **Single Shipment** button in the shopping cart.
 2. Select one shipping address and shipping method for all the items.
 3. Choose multiple payment methods.
 - **Results:** The item details area is refreshed and all the items have the same address and shipping method. The order is created successfully.
- Ship items to multiple addresses and single payment method
 - **Scenario:** In the testing, you can put three or more items in the shopping cart.
 - **Process:** The process is as follows:
 1. Click the **Multiple Shipment** button in the shopping cart.
 2. Select different shipping addresses and shipping methods for each item.
 3. Click the advanced shipping option and enter a requested shipping date and shipping instructions.
 4. Choose single payment method.
 - **Results:** The item details area is refreshed and all the items are arranged in tabular form, with each item having its own shipping address and shipping method.

The order total is updated to reflect the changed charges.

Requested shipping dates and shipping instructions are entered successfully.

The order is created successfully.

- Ship items to multiple addresses and multiple payment methods
 - **Scenario:** In the testing, you can put three or more items in the shopping cart.
 - **Process:** The process is as follows:
 1. Click the **Multiple Shipment** button in the shopping cart.
 2. Select a different shipping address and shipping method for each item.
 3. Click the advanced shipping option and enter shipping dates and shipping instructions.
 4. Choose multiple payment methods.
 - **Results:** The item details area is refreshed and all the items are arranged in tabular form, with each item having its own shipping address and shipping method drop down list.
- The order total is updated to reflect the changed charges.
- Request shipping date and shipping instruction entered are successfully.
- Order is created successfully.

► **Manage wishlist**

The shopper can add item to a wishlist from a product detail page, tooltip of a product quickview, e-marketing spot, search result, compare result, Fast Finder, scrollable pane, customer service links, shopping cart, and check-out page.

► **Compare attributes of selected items**

Items dragged to the compare zone can be compared by clicking a button in the compare zone. The comparison page loads, showing the items side-by-side with attribute name, values, and prices. The **Add to Cart** button is displayed below each item.

In the product tooltip quickview, the **Add to Compare** button adds the current item to the compare zone.

► **Faster finder**

The shopper can use Fast Finder to narrow product selections.

When a shopper performs this operation, the resulting product list narrows. An option is given to change the display details and the layout of the products displayed.

The resulting products narrow with only products in a selected price range displayed. An option to change the display details and layout of the products is displayed. A tooltip displays more information about the product.

4.1.2 Usability testing

Usability testing involves measuring the ease with which users can complete common tasks in your store. When your store has Web 2.0 features, the shopping experience should be more convenient, with greater usability than the traditional Web 1.0 store.

Goals of usability testing

Usability testing is a black-box testing technique. The aim is to observe people using the product to discover errors and areas of improvement. Usability testing generally involves measuring how well test subjects respond in four areas:

- ▶ Performance
How much time, and how many steps, are required for people to complete basic tasks? (For example, find something to buy, create a new account, and order the item)
- ▶ Accuracy
How many mistakes did people make? (And were they fatal or recoverable with the right information?)
- ▶ Recall
How much does the person remember afterwards, or after periods of non-use?
- ▶ Emotional response
How does the person feel about the tasks completed? Is the person confident, stressed? Would the user recommend this system to a friend?

The results of the first test can be treated as a base line or control measurement. All subsequent tests can then be compared to the baseline to indicate improvement.

Best practice on usability of the Web 2.0 Store

The following list details some usability aspects you should pay attention to when testing the Web 2.0 Store:

- ▶ Consistent behavior from browser to browser
Your Web site should not restrict any of the features behave differently in different browsers or in different versions of the same browser. Usually the shopper can log onto the Web 2.0 Store using any browser installed on the client machine. If one feature of your store happens to not work in a shopper's browser, it is with likely that the shopper will leave the online store and abandon any shopping carts. To avoid this, in the testing phase, ensure that all Web 2.0 Store functions behave similarly from browser to browser.

Because you do not know what browser your shopper is using, test against all main versions of each supported browser.

Dojo provides well-designed APIs and widgets shielding the complexity of browser-specific logic in AJAX code, and if you are just using Dojo APIs or widgets, you need not care about the browsers, and Dojo will handle it properly. During development of your Web 2.0 Store, avoid writing the browser-specific AJAX code yourself unless you cannot find any existing Dojo APIs or widgets that you can use. We suggest you follow this best practice because it is risky to write the browser-specific code, and the code may not work well in those browsers you failed to address in your code.

You may experience formatting problems caused by browser differences that are hard to solve because the HTML is dynamically generated. Usually, it is inherited in JavaScript, and sometimes that the version of Dojo is not the latest. The WebSphere Commerce Web 2.0 Store solution uses Dojo 1.0 (the version with a lot of enhancements), upgrading from Dojo 0.x may resolve some of these problems.

- Window resize

When the shopper is browsing your Web 2.0 Store, he can arbitrarily change the size of the window of your Web 2.0 Store. The operation of window resizing should not hurt any layout of display elements in your Web 2.0 Store. If an item's appearance becomes unpleasant to the shopper, they may leave the site and abandon the shopping cart.

For example, the problem may occur after the shopper changes to a smaller window size. The scroll bar appears, but when the shopper drags the scroll bar the information that should display on the page is not shown well. The scroll bar appears not to work. This issue conveys a message of poor design to the shopper, and heavily degrades the shopper's confidence in the site.

Another example may occur after the shopper maximizes the window. It is expected that everything displays well as usual. But in some cases, the store page will still display only the content shown before the window is maximized. This is ugly.

Test the window resizing functionality to ensure your store behaves consistently.

- Workstation resolution

Your shoppers will have a variety of screen resolutions when they browse your Web 2.0 Store. You may want to notify the shoppers regarding the best resolution to browse your site, though most shoppers are not willing to change their resolution to meet your requirements. A best practice is to select a low enough resolution for the majority of your customers screens. For a typical B2C Web site, the decision is usually split between 800x600 and 1024x768 resolutions. Which one you select is based on your audience, but typically

1024x768 is the most widely used screen size according to the following Web page:

<http://www.hobo-web.co.uk/tips/25.htm>

If you chose 1024x768 as your screen size, you should test at 800x600 to ensure that your site can support that group of shoppers. Your site should, of course, be tested to support higher screen resolutions as well.

- Friendly error message

When the shopper experiences your Web 2.0 Store, he may perform an incorrect operation or incorrectly input required information. When such an activity occurs, the request is sent to the server side, but an exception occurs and the request cannot be successfully handled on the server side. The store page should display the exact error, clearly informing the shopper what happened. If the store page does not indicate an exception, the shopper may assume everything works well and continue shopping. In some cases, continuing shopping can cause unexpected behaviors and generate out-of-control errors in the system. If the store page displays an unmeaningful message to the shopper, it may confuse the shopper and the shopper may abandon the shopping cart, because the shopper has no idea what he should do to fix the error from his side. Make the exception or error message shown to the shopper friendly. Avoid using the technical description in the error message. The shopper may appreciate a message that is clearly understood in the business domain. If you do not want to expose the exact error message to the shopper, calmly share the situation with the shopper hoping they will remain patient.

Another aspect to consider is that when an error happens, it is nice to have the store automatically forward the shopper to a page where the shopper may take action to correct the error. The page should provide specific instructions on how to resolve the error and continue shopping. This feature is convenient for the shopper and reduces the possibility of the shopper abandoning the shopping cart.

- Appropriate indicator in the drop area

The drag and drop (DnD) feature is common in the Web 2.0 Store. In previous chapters, we described what DnD is for in the WebSphere Commerce Web 2.0 Store solution and how it is implemented in typical scenarios. When the shopper drags something to a drop area, there should be some indication that the current area is an allowable drop area. In the WebSphere Commerce Web 2.0 Store solution, we use the DnD functionality provided by Dojo. If a shopper tries to drag something into a non-allowable area, a red plus indicator displays. When the dragged item is in an allowable drop area, a green plus indicator will appear, at which point the shopper is allowed to complete the activity.

If you are implementing the DnD feature in your store through Dojo DnD functionality, this is not a major concern, because Dojo can handle it if you correctly define the allowable drop area. If you are building the DnD feature yourself, or are using another Web 2.0 technology, ensure there is an easy-to-understand indication whether the mouse is in a desired drop area. Otherwise, the shopper has no idea what exact area he can release the mouse to complete the DnD operation.

- Consistent behaviors in multilingual and multi-cultural environment

If your Web 2.0 Store supports multiple languages, ensure the exact or correct information is in each supported language. For example, the store should show everything correctly when the shopper selects the double byte language (Chinese, Japanese, Korean, and so forth) as the preferred language. Your store page should show everything correctly in any supported browsers when the shopper selects a BIDI (bi-directional) language as the preferred language.

Carefully consider cultural differences, regulations, and the laws of the corresponding country.

Hallway usability testing

Hallway testing (or hallway usability testing) is a software usability testing methodology. The name of the technique, hallway, refers to the fact that the testers should be random people who pass by in the hallway. The idea behind hallway testing is that the testers who run the usability testing do not come from in-house and trained groups. For your Web 2.0 Store test, find five or six people who have not used this store previously to execute the usability testing.

To test the usability of your Web 2.0 Store, set up a usability test that involves carefully creating a scenario, or realistic situation, where the tester performs a list of tasks while observers watch and take notes. Several other test instruments (such as scripted instructions, paper prototypes, and pre- and post-test questionnaires) are used to gather feedback on the product being tested. For example, to test the function of dragging an item to the shopping cart, a scenario would describe a situation where a shopper searches for and then buys an item in a product page. The aim is to observe how people function in a realistic manner, so that developers can identify problem areas.

4.1.3 Performance testing

When your Web 2.0 Store functionality has fulfilled your requirements and it works as expected, performance testing should be executed to ensure the performance of your Web 2.0 Store.

The main purpose of the performance testing is to verify the following states:

- ▶ All functions and business scenarios either meet or exceed your requirements
- ▶ Microcosm, stress, scalability, and reliability tests meet your requirements
- ▶ (Optional) Performance is equal to, or better than, the original Web 1.0 Store

Performance test types and results

There are four performance test types, as listed below. Under each test type is listed the criterion for judging a test successful, and the recommended test results to be captured.

- ▶ Microcosm
 - Pass criterion of microcosm sequential test
 - No deadlocks in the database
 - Microcosm finishes without finding any system errors and unexpected application errors
 - Test cases run successfully for the entire test duration (A time slice (for example, five minutes) per scenarios)
 - Pass criterion of microcosm concurrent test
 - Do not exceed the max deadlock number (for example, 10)
 - Do not exceed the max error rate (for example, 1%)
 - Test cases run successfully for the entire test duration (for example, one hour)
 - Recommended test results to be captured
 - Test result report (including test machines information and tuning parameters, and so forth.)
 - Baseline Report and Time Series Data (for example, using SilkPerformer)
 - CPU usage on all nodes
 - Disk I/O on database node
 - Deadlock report
 - JVM™ logs on the application node

- ▶ Stress
 - Pass criterion
 - Do not exceed the maximum error rate for transactions (for example, 1%)
 - Do not exceed the maximum error rate for Web hits (for example, 0.1%)
 - Numbers of users have reached its target
 - Throughput reaches its target as defined for the individual test scenarios
 - No visible throughput degradation
 - Test scenario runs successfully for a long time (for example, three hours)
 - Recommended test results to be captured
 - Test result report (including test machines information and tuning parameters, and so forth)
 - Baseline Report and Time Series Data (for example, using SilkPerformer)
 - CPU usage on all nodes
 - Disk I/O on database node
 - Deadlock report
 - JVM logs on the application node
- ▶ Scalability
 - Pass criterion
 - Do not exceed the maximum error rate for transactions (for example, 1%)
 - Do not exceed the maximum error rate for Web hits (for example, 0.1%)
 - Response time is less than target, on average, for each server function that is executed (for example, six seconds)
 - Test scenario runs successfully for a long time (for example, three hours)

- Recommended test results to be captured
 - Test result report (including test machines information and tuning parameters, and so forth.)
 - Baseline Report and Time Series Data (for example, using SilkPerformer)
 - Database logs on database node (for example, db2 snapshots, db2diag.log)
- ▶ Reliability
 - Pass criterion
 - Do not exceed the maximum error rate for transactions (for example, 1%)
 - Response time is less than target, on average, for each server function that is executed (for example, six seconds)
 - No memory leak or heap fragmentation problem found
 - Test scenario runs successfully for long time (for example, 72 hours)
 - Recommended test results to be captured
 - Test result report (including test machines information and tuning parameters, and so forth.)
 - Baseline Report and Time Series Data (for example, using SilkPerformer)
 - CPU usage on all nodes
 - Disk I/O on database node
 - JVM logs on the application node
 - Starting time, size, and execution time for utilities (for example, DBclean, Cache invalidation, and ETL, and so forth)

Test data

The acceptance test can be run based on relatively small-sized data to reduce the risk that a functional problem breaks the performance test. The following list details the small-size sample data for the Web 2.0 Store acceptance test.

- ▶ 500 registered users
- ▶ 5 first categories
- ▶ 4 second categories
- ▶ 10 products
- ▶ 1 item

If the acceptance test passes, a more formal performance test can be run in the Web 2.0 Store.

The following list details the large-size sample data in the Web 2.0 Store test.

- ▶ **Member**
 - Belong to the same buyer organization that has one default business account
 - Each registered user has four orders for a total of 170,000 completed orders, 20,000 pending orders, and 10,000 canceled orders for registered users. Each order contains three order items.
 - Each registered user has two addresses
 - 150,000 guest buyers with a total of 85,000 completed orders, 10,000 pending orders, and 5,000 canceled orders for the first 100,000 guest users. Each guest user should have no more than one order and the other guests has none at all. Each order contains two order items.
 - Each guest user has two address.
- ▶ **Catalog**
 - 10 top-level categories (50,000 catalog entries)
 - 20 second-level categories
 - 25 three-level categories
 - 10 products
 - 1 on hand item (quantity=1,000,000)

Web 2.0 Store test scenario best practice

This section details common potential performance risks of a store with Web 2.0 features, and presents a sample performance verification test scenario for your reference.

Potential Web 2.0 Store performance risks

The following list details potential Web 2.0 Store performance risks:

- ▶ Because of the use of AJAX, asynchronous execution of many JSPs for a given user at the same time may cause deadlocks. These deadlocks would not be detected.
- ▶ The assumption that the response time of an individual weblet is independent of the response time of another weblet may not be true.
- ▶ The assumption that we can scale down the workload (to ensure that CPU does not reach 100%) would still result in valid response time testing may not be true.

- ▶ The assumption that the WebSphere Commerce server operates on a four-way CPU and threads may execute simultaneously.
- ▶ Execute a task to delete an item from shopping cart results in the following asynchronous JSP calls:
 - Refresh mini shopping cart contents
 - Update cart total in the header
 - Display eSpot
- ▶ Simultaneous execution of JSPs in different threads for the same user going against the same database may result in deadlocks.

Sample test scenario

Table 4-1 on page 209 shows the sample Web 2.0 Store performance verification test scenarios.

These sample test scenarios are organized by shopper type. The first column listed three shopper types used in the testing:

- ▶ Registered Shopper
- ▶ Guest
- ▶ New Registered Shopper

The second column lists the scenario name and the third column lists the rate of each scenario. The last column list the Detail flow of each scenario.

Table 4-1 Sample Web 2.0 Store performance testing scenarios

Shopper type ^a	Scenario Name	Rate ^b	Detail flow ^c
Registered Shopper (20%)	Browse	70%	Store Front → Log on → Browse
	Add to Shopping Cart	5%	Store Front → Log on → Browse → Add to Shopping cart (Click the button to add to shopping cart - 50%) (Drag and drop - 50%)
	Process Order	5%	Store Front → Log on → Browse → Add to Shopping cart → Shopping Cart Page → Process Order (Click the button to add to shopping cart - 50%) (Drag and drop - 50%) (Click shipping address - 50%) (Current shipping address - 50%)
	Add to Wish List	5%	Store Front → Log on → Browse → Add to Wish List (Click the button to add to wishlist - 50%) (Drag and drop - 50%)
	Search Product	15%	Store Front → Log on → Search Product
Guest (60%)	Browse	70%	Store Front → Browse
	Process Order	5%	Store Front → Browse → Add to Shopping cart → Shopping Cart Page → Filling Address → Process Order
	Add To Shopping Cart	5%	Store Front → Log on → Browse → Add to Shopping Cart
	Compare	5%	Store Front → Browse → Compare
	Search Product	15%	Store Front → Search Product
New Registered Shopper (20%)	Browse	70%	Store Front → Register → Browse
	Add To Shopping Cart	5%	Store Front → Register → Browse → Add to Shopping Cart (Click the button to add to shopping cart - 50%) (Drag and drop - 50%)
	Process Order	10%	Store Front → Register → Browse → Add to Shopping Cart → Shopping Cart Page → Process Order (Click the button to add to shopping cart - 50%) (Drag and drop - 50%) (Click shipping address - 50%) (Current shipping address - 50%)
	Search Product	15%	Store Front → Register → Search Product

- a. The percentage listed in this column is the percentage of this shopper type in all of the shoppers in the testing scenarios.
- b. This number represents the percentage that this scenario is found in all of the scenarios of the current shopper type.
- c. The information in parentheses represents the frequency with which the shopper performs that action.

4.2 Debugging your Web 2.0 Store

Debugging a Web 2.0 Store is not much different than debugging a Web 1.0 Store. The key to any type of problem determination is understanding the behavior and interactions expected by the system, then breaking down and isolating the problem.

This section summarizes the Web 2.0 interactions that take place in a typical WebSphere Commerce store. It provides a model where you can easily isolate problem areas to solve what appears to be a complex problem.

4.2.1 Web 2.0 interactions

A Web 2.0 Store has an increased level of complexity compared to a traditional store when it comes to the interactions between the client and server. Figure 4-1 on page 211 illustrates the general flow of interactions in a Web 2.0 Store.

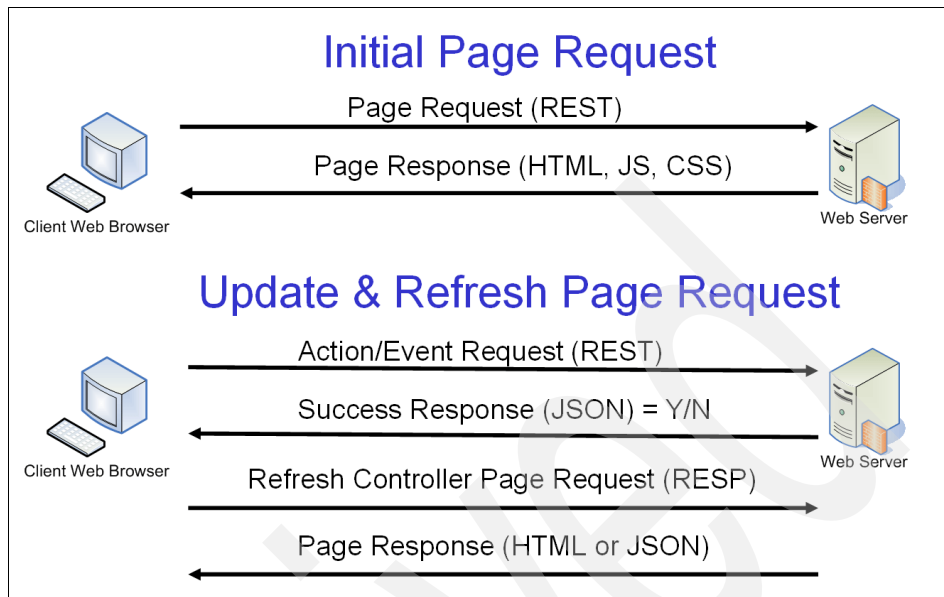


Figure 4-1 General flow of interactions in a Web 2.0 Store

The Initial Page Request is what we are used to in a Web 1.0 store. It is the new set of interactions in the Update and Refresh section that we want to focus on for Web 2.0.

4.2.2 Breaking down the possible problem areas

Using the interaction model shown in Figure 4-1, you can easily see that there are interactions created by the client's Web browser, and that the response or data is generated and returned by the WebSphere Commerce server. The client itself is responsible for making the necessary requests to the WebSphere Commerce server and to populate and consume the responses by the WebSphere Commerce server. This is where Web 2.0 can have problems.

The server processes the requests the same as in Web 1.0. Therefore, the techniques to debug these server-side problems are the same. However, the difficult part in a Web 2.0 Store is understanding where something has broken. Any of the steps in the interaction can fail (server or client), but the visible result for the user does not make it clear as to which step has the problem.

4.2.3 Tools to debug and isolate problems

The interaction model shows three areas to analyze and solve a problem:

- ▶ Client
- ▶ Server
- ▶ Interactions between the client and server.

Client-side debugging

Client-side debugging using the Dojo toolkit adds an extra dimension of complexity. You can add traces to your JSP files, just like in a Web 1.0 scenario. However, aside from using your regular JavaScript debugging techniques, you also need to enable Dojo debugging on the client. The Dojo toolkit provides its own debugging framework. When enabled, it dynamically adds traces to the bottom of your HTML pages. Dojo debugging helps you troubleshoot why dynamic actions are not initializing properly, or behaving differently than expected. For example, if a product reveal does not display when mousing over a product, the Dojo debugging statements help you isolate the issue. It also helps monitor client-side events, such as a request being sent, and a response triggering a client-side event.

To enable Dojo debug tracing, open the `<Stores>\<Web20Store>\include\JSTLEnvironmentSetup.jspf` file and set `dojoDebugFlag` to true as shown in the following code:

```
<c:set var="dojoDebugFlag" value="true"/>
```

By changing a .jspx file, you need to force all the JSPs to re-compile by deleting the compiled files from the following code:

```
<WAS_profile_dir>\temp\localhost\server1\WC\Stores.war\<Web20Store>
```

Server-side debugging

This is the same type of debugging that you perform in a Web 1.0 store. Enabling `WC_SERVER` tracing helps you understand the requests that have been sent to the WebSphere Commerce server. There are also specific WebSphere Commerce component traces that can help debug issues that occur on the server side. For example, if the Add to the Cart step fails, enable the `WC_ORDER` trace to determine the cause of the failure.

Interaction between the client and server

To monitor the interactions, enable `WC_SERVER` tracing to understand whether or not a request was made by the client. You can also validate the data in the request. There are third-party tools or plug-ins that you can download to monitor the traffic at the client side.

Firebug is a plug-in to Firefox which can allow you to do this (and more). Using Firebug, you can easily track requests and responses between a client and a server. Firebug can also assist in detecting client side code defects and errors.

Firebug is available from the following Web page:

<https://addons.mozilla.org/en-US/firefox/addon/1843>

Figure 4-2 shows two requests that represent the interaction flow expected in the Web 2.0 Store.



Figure 4-2 AJAX requests to the Commerce server

You can expand the details of each request to inspect the parameters in the request. Firebug allows you to see the response data. Figure 4-3 shows the data for the AjaxInterestItemAdd request, including the request parameters.

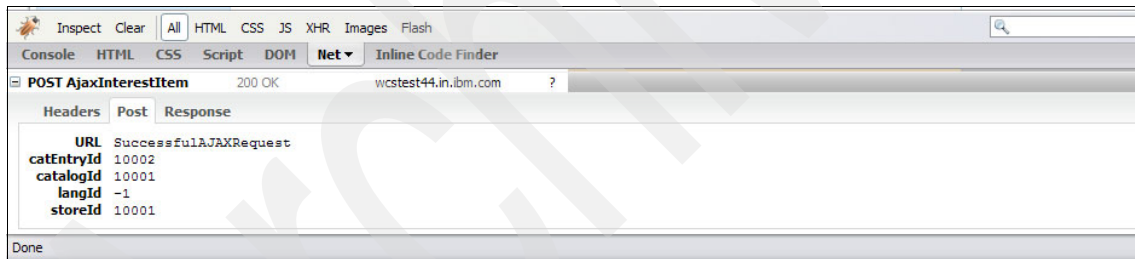


Figure 4-3 Requesting information of AjaxInterestItemAdd

Figure 4-4 shows the response corresponding to the `AjaxInterestItemAdd` request.

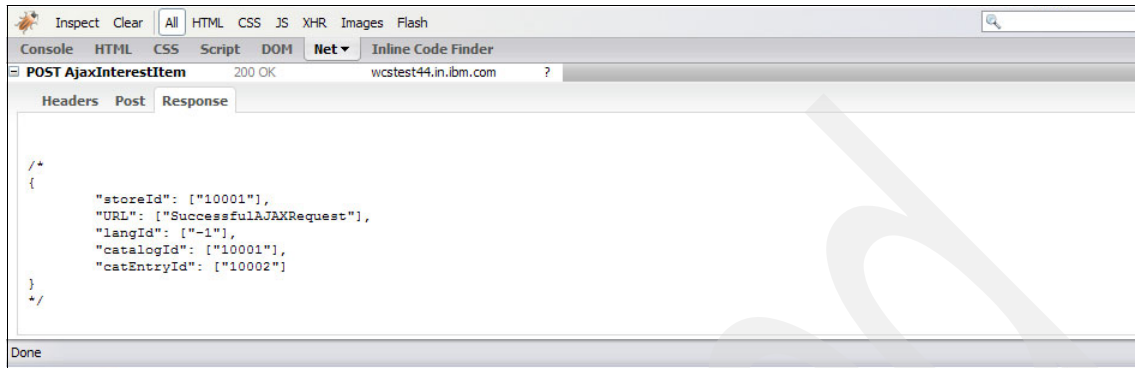


Figure 4-4 Response information of `AjaxInterestItemAdd`

You can also use Firebug to inspect a page, allowing you to understand the elements that make up a page and to identify the code that may be causing a problem when rendering. Figure 4-5 on page 215 shows an example of inspecting a section of a page. The corresponding parts on the page and in the code are highlighted so that you can determine the HTML code that is rendering the content.

Figure 4-5 is an example of inspecting a section of a page to determine the HTML code that is rendering content.

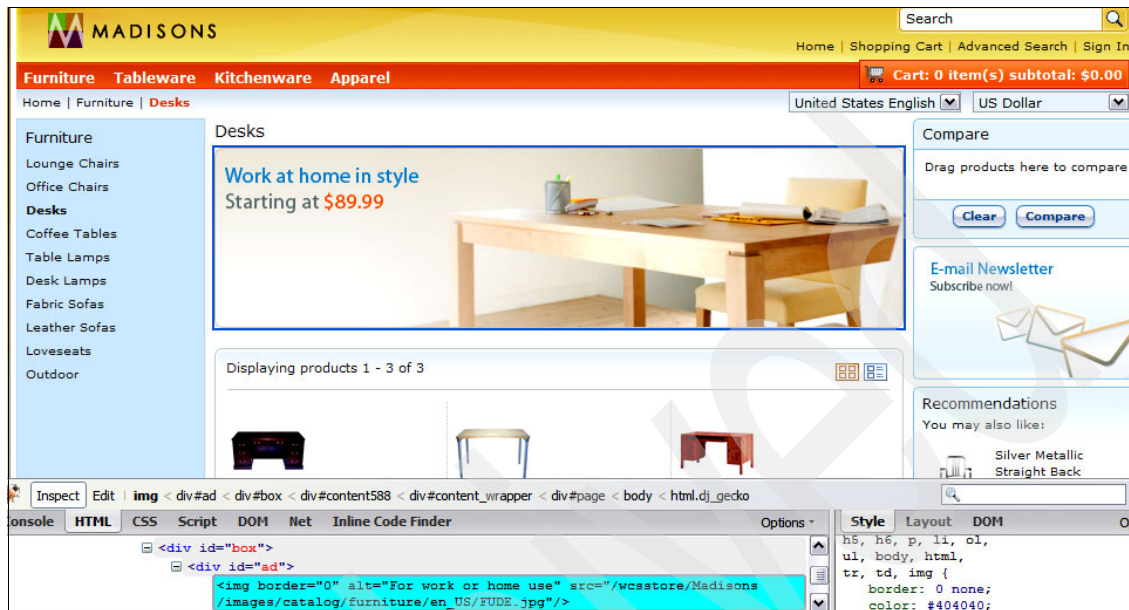


Figure 4-5 Inspecting a section of a page

4.2.4 Identifying and isolating the problem area

To help with isolating and identifying a Web 2.0 problem, you can monitor the client-server interaction. By reviewing the interactions, you can see whether the interaction that you expect actually occurred and whether the data within the interaction was correct. Let us take an example of a DnD into the mini shopping cart. You expect that once the product has been dropped into the mini shopping cart, the item is added and the mini shopping cart will reflect this change.

Looking at the interaction model, you can quickly determine where a problem may have occurred. The first interaction is a request from the client to the server, calling a command to add the item to the cart. By monitoring the interaction flow, you can isolate the problem by answering the following questions:

- Did the interaction occur?

If you do not see the request going to the server, the problem is confined to the client code. In the case of the DnD example, analyze the drop zone and the JavaScript code that executes the commands. In this case, enabling Dojo client-side debugging will help you diagnose the problem.

- Did the request have the correct data?

Did the client send the correct data to the server? If the wrong data is sent to the server, the problem is in the client because the server cannot return the correct results given the wrong inputs. Go back to the client code and validate the logic that sends or populates the parameters. In this case, Firebug or similar technology will help you diagnose which parameters were sent in the request. If the request got to the server, the application WC_SERVER trace log will contain the request parameters.

- Did the response have the correct data?

Assuming the request has been sent correctly, with the correct request parameters, did the server respond with the expected response? If not, there is an issue on the server side. Debug this like any client and server interaction similar to a Web 1.0 store interaction by checking the server logs, input and output definitions for the action requested, and request and response parameters.

- Did the result get consumed by the client?

If the server has sent the correct data back to the client, did the client act on this response? If all the steps above are working, but you still have unexpected results in the client, there is a client-side problem related to the rendering of a response. Analyze the widgets that are defined to consume the response, and whether they are set up and configured correctly. Enabling Dojo client-side debugging may help you diagnose the problem.

In the interaction model, after the original command response is received, the Web 2.0 model generally has a second request and response flow to render any updates. When reviewing the questions above, if the second request does not occur, this means that there is a problem in one of the following areas:

- In the “Did the result get consumed by the client?” question of the first request, the result did not get consumed successfully by the client.
- In the “Did the interaction occur?” question of the second request, the new request for updates did not go to the server.

In reality, both are one and the same. In both cases, the issue is confined to the client, the interaction between the first response, and the creation of the second request.

4.2.5 The divide and conquer method

When you have isolated and identified the problem area, you can recursively apply the algorithm to further focus on the problem.

Going back to our DnD example, let us assume the DnD does not create a request (for example, failure of the “Did the interaction occur?” question above). The expected behavior is that you can drag an item into the mini shopping cart, the drop zone detects the drop, and the drop zone passes the required parameters into a JavaScript function. This function executes a service to handle this event. The service calls an URL on the server. You can ask the same questions to narrow down why the DnD failed on the client side:

- ▶ Did the interaction occur?

Did the drop zone detect the item was dropped? The Dojo debug code can help identify this. If the drop zone did not detect the interaction, review your drop zone to ensure that it was defined correctly. Review your object to ensure that it can be dragged.

- ▶ Did the request have the right data?

When the drop is detected, were the right parameters passed in? If they are not correct, review how the object you are dragging is defined, or the way the widget is gathering the parameters. The Dojo debug may help.

- ▶ Did the service get invoked?

If the service is not invoked, there is a missing relationship between the drop zone event handler and the service defined to handle the event. Review both to ensure that the event handler is invoking the service, and whether the service is defined correctly.

To determine the answer to some of these questions, you may need to enable client side traces, such as the Dojo debug, to facilitate the debugging of your JSPs and JavaScript code.

The idea behind debugging methodology is to understand the interactions at a high level, then divide the flow into sections, and use clues in the request and response data to isolate the problem area. Iteratively, you can use this model to focus on a problem area and figure out the root cause.

4.2.6 Debugging a sample problem

The following section illustrates how the debugging methodology can be applied to a real-life problem. This example uses the steps above to identify the component that failed, and allows you to narrow down the problem area.

Quick Cart refresh

In this scenario, when adding items to the cart through a DnD, the quickcart section of the page does not refresh as expected. In addition, there is a Communication Error, as shown in Figure 4-6:

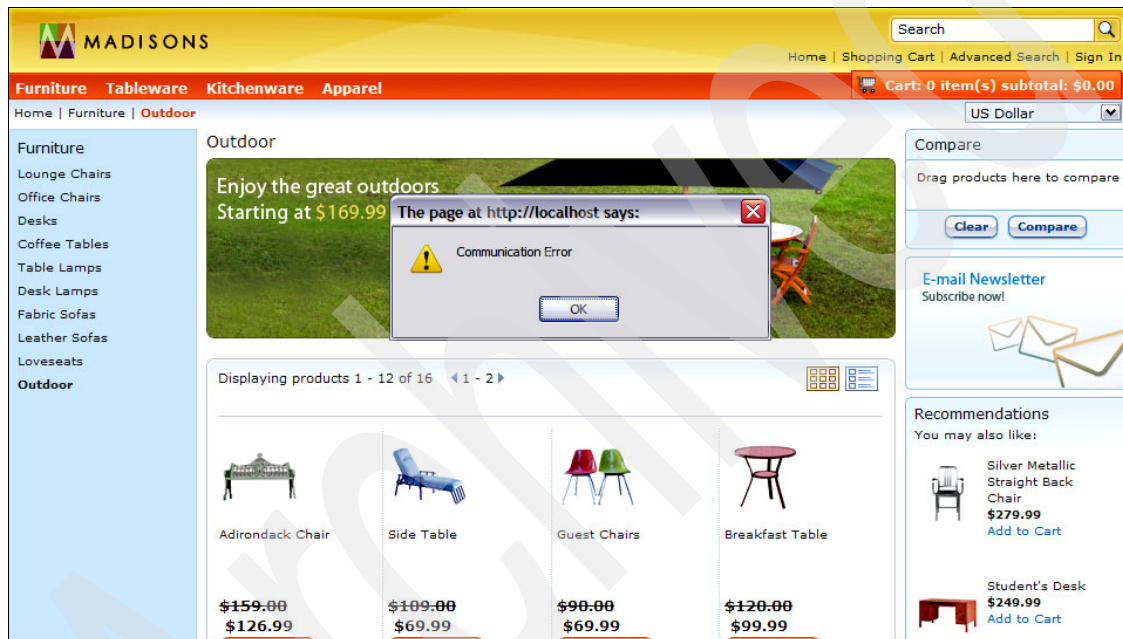


Figure 4-6 Error page when adding to a shopping cart

Reviewing the SystemOut.log indicates no errors and no sign of failures at the Application Server level. Based on this, there are no easily identifiable problems on the Commerce server. In addition, the error within the Web browser is insufficient to determine the root cause.

Without a debugging methodology, it may be difficult to understand whether you need to debug a problem on the browser (for example, Dojo, JavaScript, and so forth) or on the server. One may have started to try to debug this in the client, because it had returned a communication error.

Debugging the initial page request

To debug the error shown in Figure 4-6 on page 218, use Firebug and the debugging methodology introduced above.

When we reproduce the issue by dragging one of the items, we can observe the interactions that occur between the client and server.

Figure 4-7 outlines the set of requests that are generated by the DnD.

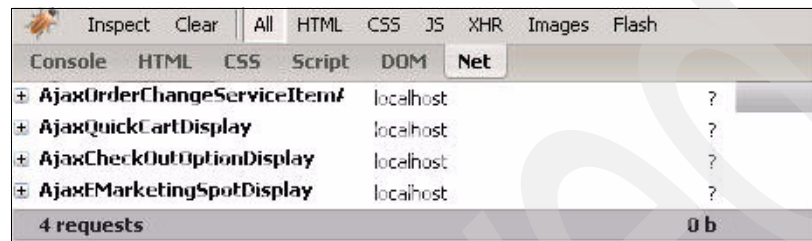


Figure 4-7 Requests to the server

Reviewing the interactions expected in this scenario, `AjaxOrderChangeServiceItemAdd` is the request made when the item is dropped into the quickcart. This maps to the Action/Event Request discussed in the interaction model. The other three requests are related to different parts of the page that refresh when the `AjaxOrderChangeServiceItemAdd` request responds successfully. In this case, the `AjaxQuickCartDisplay` request is responsible for refreshing the contents of the quickcart.

To isolate the problem, use the data from Firebug to implement the methodology introduced earlier. Inspecting the results in Firebug, we can answer the “Did the interaction occur?” question.

We can tell that the `AjaxOrderChangeServiceItemAdd` URL was called when we dragged and dropped the item into the quickcart. By clicking the + symbol to the left of the `AjaxOrderChangeServiceItemAdd` request (Figure 4-8), we can see the parameters in the request we made. Hovering the mouse on the request will give us the exact URL called.

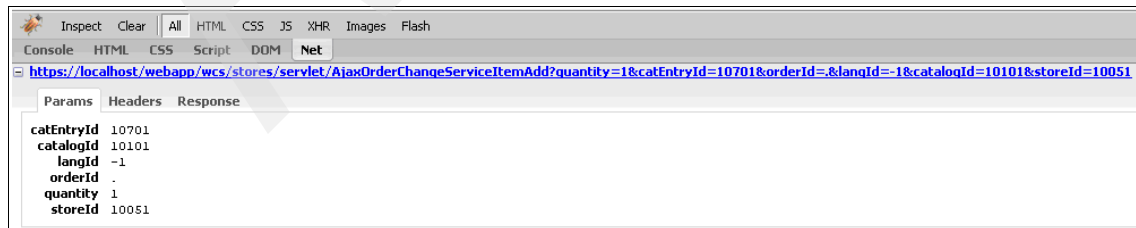


Figure 4-8 Detailed information of `AjaxOrderChangeServiceItemAdd` service

Inspecting the data in Figure 4-8 on page 219, we can be confident that the correct URL is called when we drag the item into the quickcart, and that it is being requested with the correct parameters. This answers the “Did the request have the correct data?” question in the debugging methodology.

We have easily confirmed that the DnD in the client has worked as expected, and the problem is not in the client at this step.

Inspect the response data by clicking the Response tab, as shown in Figure 4-9.

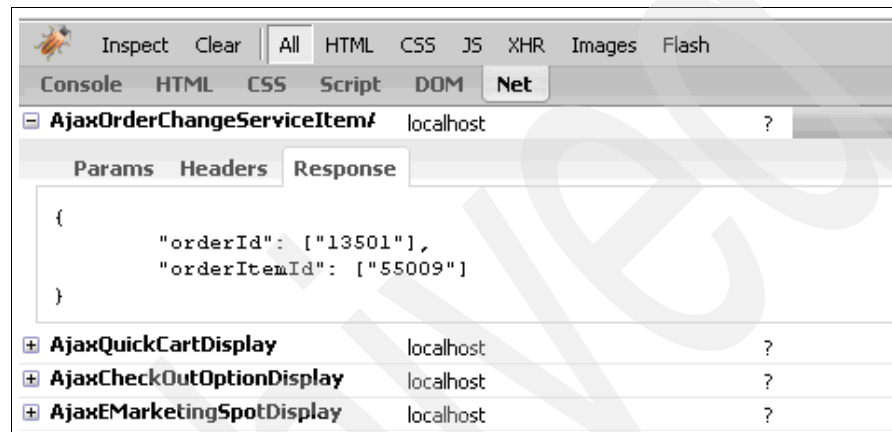


Figure 4-9 Response data

Based on this data, we can answer the “Did the response have the correct data?” question in the debugging methodology.

We can see from the output that the response was successful. This confirms that the problem did not occur on the server at this stage.

The next step in the interaction flow would be the client acknowledging this response, and making a new request to the server which refreshes the quickcart. This is the “Did the result get consumed by the client?” question in the debugging methodology.

The AjaxQuickCartDisplay request that we see in Firebug is the RefreshController Page Request in the interaction flow diagram in Figure 4-1 on page 211. The fact that we can see this in Firebug confirms that the result was consumed by the client.

We can now analyze the refresh interactions.

Debugging the refresh page request

As we did in the initial request, we can use Firebug to analyze the refresh page request.

We can answer the question as to whether the refresh request is using the correct parameters by clicking on the + symbol next to the AjaxQuickCartDisplay request, and selecting the Params tab, as shown in Figure 4-10.

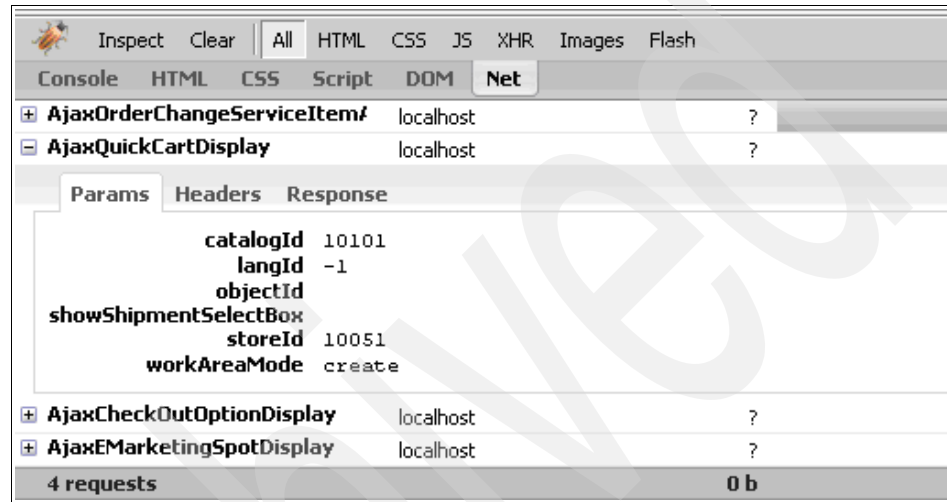


Figure 4-10 Detailed information of AjaxQuickCartDisplay request

Based on the data in this request, it appears to be correct which answers the “Did the request have the correct data?” question of the debugging methodology. The issue is not in the client at this step.

Moving onto the next step of the methodology, eviue the response data for the AjaxQuickCartDisplay request by clicking the Response tab, as shown in Figure 4-11 on page 222.

4.3 Optimizing Web 2.0 Store

Web 2.0 development introduces new performance complexity. It is important that JSP developers stay in tune with site architecture more so than ever before. The original intention of AJAX applications was to make requests smaller and quicker, which holds true. However, application designers are creating widgets that interact more frequently with the server. Requests are smaller, but they happen more often and response times are expected to be fast. Furthermore, dynamic interfaces require increased processing power from client-side browsers. The Web 2.0 performance optimization techniques focus more on front-end development, where all the rules and principals for developing Web 1.0 applications still apply. Figure 4-28 outlines some issues to consider when developing Web 2.0 storefronts. As a best practice, optimization should be thought of at every level in your site architecture.

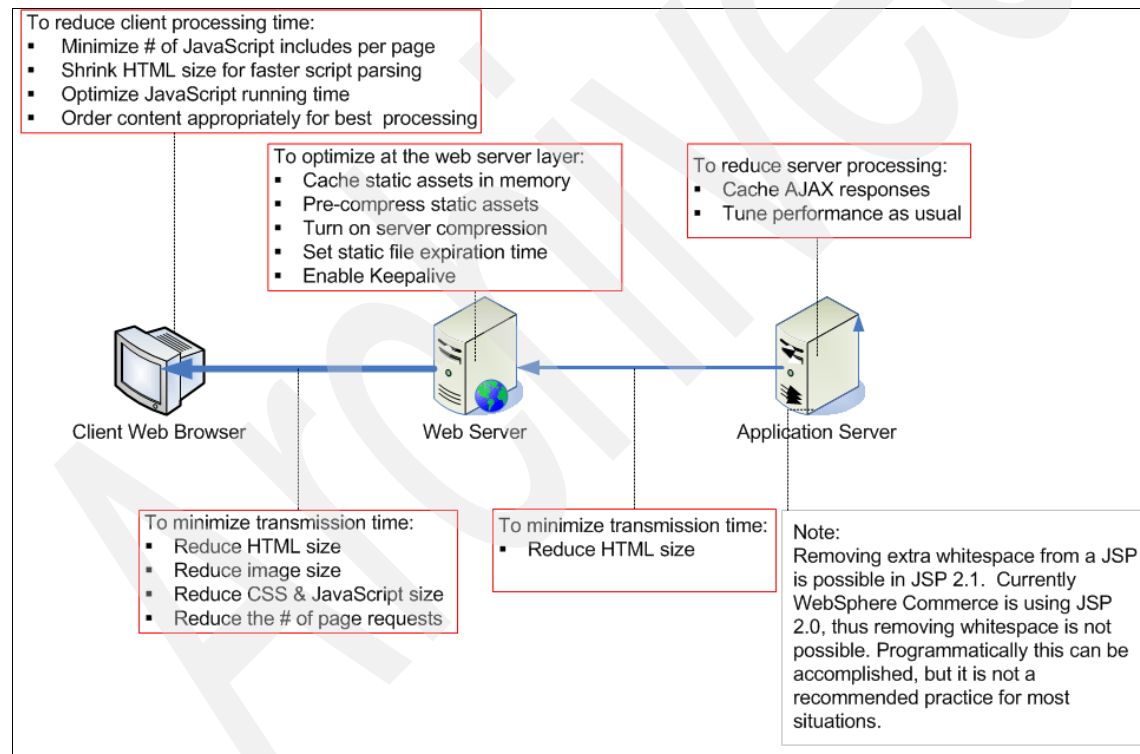


Figure 4-13 Web 2.0 optimization outline

This section consists of four parts:

- ▶ Reducing page weight by profiling Dojo code, reducing download file sizes, and compressing data.
- ▶ General performance tips to use as a project reference.
- ▶ Advanced UI performance tips for specialized Web developers.
- ▶ Efficient handling of requests on the client side.

4.3.1 Page weight review and performance tips

Inspecting and optimizing the weight of a page is important for site performance and is simple using tools such as Firebug or Tamper Data. The goal is to ensure that page size is relatively small. 300 KB to 400 KB for all downloaded content on a home page is a good average to follow. Aim for HTML to be 20 KB or less once compressed. In this section, we use Firebug to review page weight. If you use Table 4-2 as a reference of page size to client wait time, and the weight of a page is 400 KB on a fast cable modem 1 Mbps, the page will take approximately three seconds to load. With a dial-up connection, a 400 KB page may take 50 seconds to load. Initial loads are always slower than subsequent loads, because Web browsers cache content and do not reload it unless needed.

Table 4-2 *Page weight*

Connection speed	20 KB Page Size	40 KB Page Size	100 KB Page Size
56 Kbps	2 sec	5 sec	13 sec
64 Kbps	2 sec	2 sec	12 sec
128 Kbps	1 sec	2 sec	6 sec
256 Kbps (DSL/Cable)	<1 sec	1 sec	3 sec

File size:

1 KB (one KiloByte) = 1,024 Bytes (approximately 1 thousand Bytes)

Data transfer size:

1 kbps = 1,000 bits per second

It has been shown in various studies that long wait times cause your audience to abandon your site. According to Marketing Experiments, half of your visitors will give up waiting after a 15 second wait time.

See the following Web page for more information about Marketing Experiments and page weighting:

<http://www.marketingexperiments.com/improving-website-conversion/page-weight.html>

As a best practice, you should target a five seconds wait time for a complex page, and less for all other pages. Always keep in mind on average, the faster your pages load, the more pages a shopper will visit, which gives you ample time to push more creative marketing content to them.

Looking at the Madisons store in Figure 4-14, you can see that there are 81 file requests just on the homepage of the Madisons store. Looking closely, you will notice that a lot of these requests are made for Dojo-specific JavaScript files. Dojo requests files dynamically as they are needed. Trips to and from the server are costly, so we create a custom dojo.js file to contain all the code your application needs to perform its actions. The end result is a larger dojo.js file, but the number of subsequent file downloads are minimized.

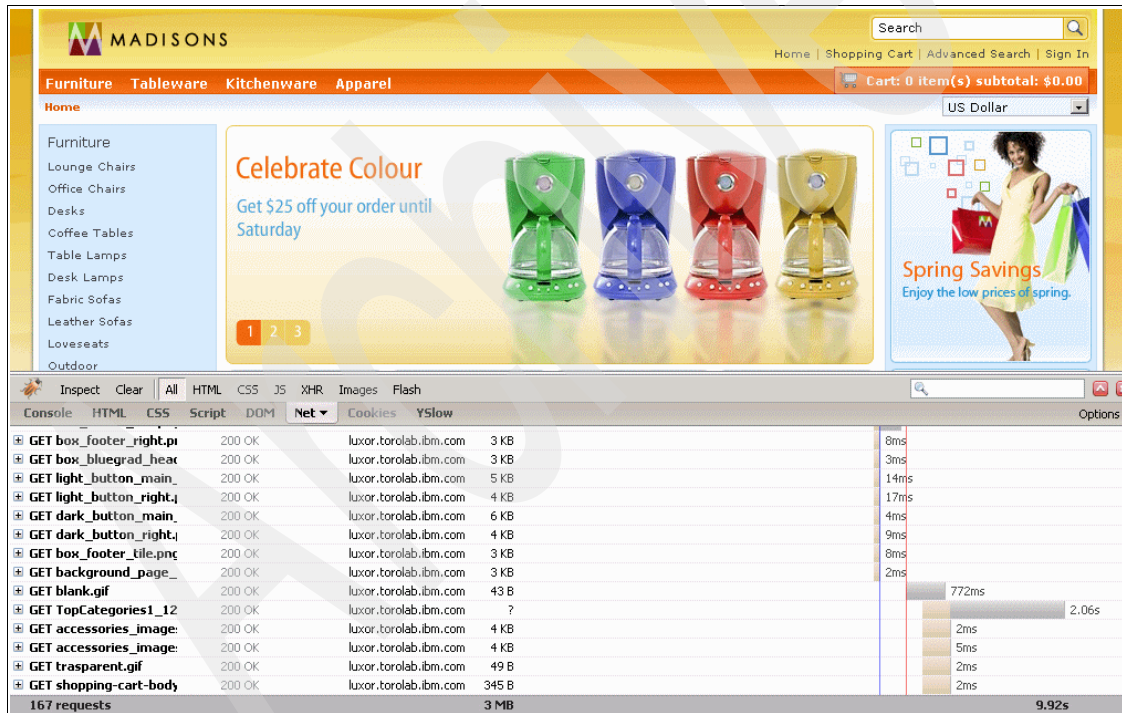


Figure 4-14 File requests in the home page of the Madisons store

There are two key factors to keep in mind when reading the Firebug timeline:

- ▶ JavaScript file download time delay

If you look at the JavaScript files downloaded in Firebug (Figure 4-15), notice that each file waits for the previous one to finish downloading, which causes a huge time delay. If all your JavaScript includes are in the header of your JSP, the resulting request will leave your visitors page blank until all the files are loaded. The ideal behavior is to have minimal file includes in the header and to cache them on the client's Web browser. Creating a custom Dojo file drastically minimizes the number of file downloads which is important.

- ▶ Static file loads from the cache

A dark gray timeline means the file was downloaded and a light gray timeline means it was loaded from cache. See Figure 4-15. On a production system, we want all static files loaded from cache on the second request. If a file is downloaded and cached by a Web browser, the next time it requests the file, a file request with the If-Modified-Since header is sent to the Web server. If the file has not been changed, the Web server responds back with a 304 Not Modified response and the Web browser uses the file it has in cache.

Console	HTML	CSS	Script	DOM	Net	Inline Code Finder
+	GET CategoryDisplay.js	304 Not Modified	localhost	65 KB	313ms	
+	GET ErrorHandler.js	304 Not Modified	localhost	14 KB	94ms	
+	GET ServicesDeclaration	304 Not Modified	localhost	14 KB	63ms	
+	GET StoreCommonUtiliti	304 Not Modified	localhost	12 KB	16ms	
+	GET ServicesEventMapp	304 Not Modified	localhost	2 KB	16ms	
+	GET CommonContextsDi	304 Not Modified	localhost	4 KB	15ms	
+	GET CommonControllers	304 Not Modified	localhost	10 KB	16ms	
+	GET CatalogEntryThumb	304 Not Modified	localhost	19 KB	32ms	
8 requests					137 KB (137 KB from cache)	

Figure 4-15 JavaScript files downloaded on home page

The benefit of minimizing the number of downloads and the size of them is important. The following list details tasks to perform after code development, when you are building your source for deployment to your application server and Web server:

- ▶ Create a dojo.js build.
- ▶ Run CSSTidy on your cascading style sheets.
- ▶ Run ShrinkSafe on your JavaScript files.
- ▶ (Optional) Pre-compress your static files.

Creating a custom dojo.js file gathers all Dojo-related files in one single JavaScript file and compacts it. This function can be incorporated into the build process, or can be done by a developer and checked into a source control repository. If you are customizing the Dojo library files, as a developer you want to use the standard dojo.js file, which is approximately only 4 KB in size. After

creating a dojo.js custom build, the file should contain all your source code and be substantially larger on your production system. To create a custom distribution, see the Dojo documentation article “Creating a Custom Distribution” available at the following Web page:

<http://manual.dojotoolkit.org/book/dojo-book-0-4/part-6-customizing-dojo-builds-better-performance/dojo-build-system/creating-cust>

The Madisons store is shipped with a default Madisons.profile.js file that defines all the packages to build into your custom distribution.

When you have successfully built your new distribution, the best practice is to place your new dojo.js file on both the application server and the Web server. Even though your new distribution file contains all the code your application may need, you should still copy all your Dojo files to both the application server and Web server.

Figure 4-16 on page 228 shows the results of the Madisons home page given that a custom dojo.js distribution file was created. Notice that the homepage now makes 20 requests instead of 81. Approximately 80% of the user response time is spent on the client side, reducing the number of subsequent requests, resulting in a substantial performance boost.

The steps for you to create a custom distribution are as follows:

1. Open Windows Explorer, and change to the target directory (We call it <Optimization>).
2. Open the following file in a text editor (UltraEdit):
`<Optimization>\Dojo101-profile\buildscripts\profiles\madisons2.profile.js`
This file lists all the Dojo packages that are built into your new Dojo distribution. Minimize this list so your new Dojo package is small. If you leave out a file you need, Dojo will download the file. So, if there are some packages that are rarely used, you may not want to include them in this distribution.
3. When the custom distribution is built, you will find it in the following directory:
`<Optimization>\Dojo101-profile\release.`
4. Copy the WebSphere Commerce widgets to the Dojo build package so that they can be included in the custom Dojo distribution for your store. Copy the entire wc folder from <Stores>\dojo101 to <Optimization>\Dojo101-profile.
5. Open a DOS prompt and navigate to the
`<Optimization>\Dojo101-profile\buildscripts` directory.

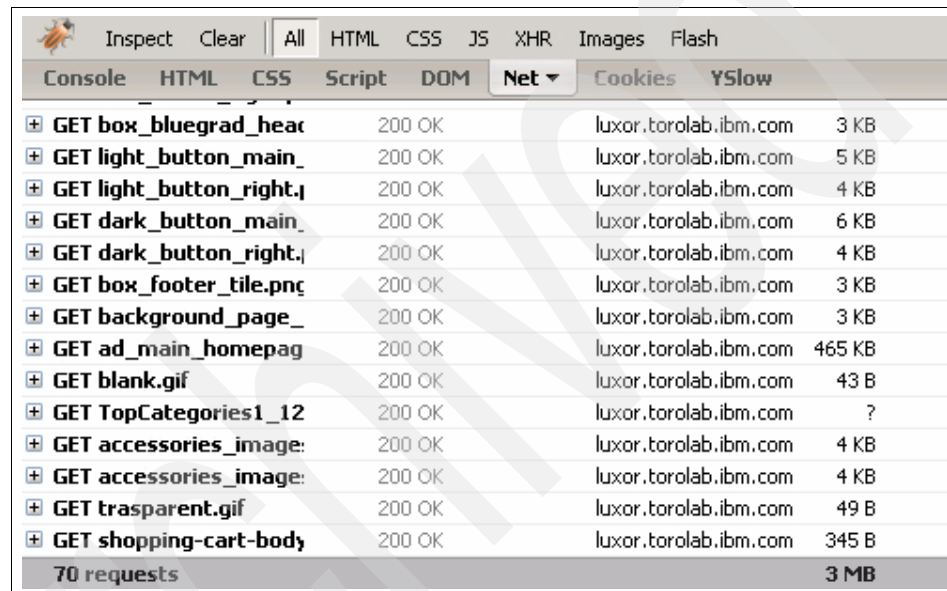
Run the following command:

```
ant -DprofileFile=profiles/madisons2.profile.js clean release  
intern-strings
```

When finished, you will notice a new directory called:
<Optimization>\Dojo101-profile\release\dojo

The correct procedure is to use the whole folder when running in production. For now, we are going to copy the
<Optimization>\Dojo101-profile\release\dojo\dojo.js file to
<Stores>\dojo101\dojo.js (replaces the file that is there).

You will notice that the homepage now only makes 20 requests instead of 81. This is 4.05 times less. See Figure 4-16.



The screenshot shows the Firefox Developer Tools Network tab with the 'Net' dropdown menu open. The table lists 15 requests, all with a status of '200 OK' and a response from 'luxor.torolab.ibm.com'. The total number of requests is 70, and the total size is 3 MB.

Request	Status	Response	Size
GET box_bluegrad_head	200 OK	luxor.torolab.ibm.com	3 KB
GET light_button_main	200 OK	luxor.torolab.ibm.com	5 KB
GET light_button_right	200 OK	luxor.torolab.ibm.com	4 KB
GET dark_button_main	200 OK	luxor.torolab.ibm.com	6 KB
GET dark_button_right	200 OK	luxor.torolab.ibm.com	4 KB
GET box_footer_tile.png	200 OK	luxor.torolab.ibm.com	3 KB
GET background_page	200 OK	luxor.torolab.ibm.com	3 KB
GET ad_main_homepag	200 OK	luxor.torolab.ibm.com	465 KB
GET blank.gif	200 OK	luxor.torolab.ibm.com	43 B
GET TopCategories1_12	200 OK	luxor.torolab.ibm.com	?
GET accessories_image	200 OK	luxor.torolab.ibm.com	4 KB
GET accessories_image	200 OK	luxor.torolab.ibm.com	4 KB
GET transparent.gif	200 OK	luxor.torolab.ibm.com	49 B
GET shopping-cart-body	200 OK	luxor.torolab.ibm.com	345 B
70 requests			3 MB

Figure 4-16 Requests after optimization

A helpful tool to analyze Web performance of your application is called YSlow. This utility gives you hints and tips on optimizing each page on your Web site. It is available from the following Web page:

<http://developer.yahoo.com/yslow>

Another key aspect of optimizing response time on the client side is to shrink the size of the files sent down the wire. Two tools that will help you do this reliably are CSSTidy and ShrinkSafe. Both tools should be run when building your application for deployment. CSSTidy compacts your cascading style sheets and

ShrinkSafe minimizes your JavaScript files. These tools may single out programmatic errors or bad conventions in your code to help you improve the reliability of your application.

► CSSTidy

- Compacts your cascading style sheets.

- Example usage:

```
csstidy Master1_1.css Master1_1-2.css
```

- Available from the following Web page:

<http://csstidy.sourceforge.net/>

► ShrinkSafe

- Shrinks JavaScript files

- Example usage:

```
java -jar custom_rhino.jar -opt -1 -c StoreAccordion.js >  
StoreAccordion2.js 2>&1
```

- Available from the following Web page:

<http://dojotoolkit.org/docs/shrinksafe>

In this example, StoreAccordion.js is shrunk from 50 KB to 20 KB.

CSSTidy and ShrinkSafe will compact your static assets. However, at your applications Web server tier, you will still want to enable compression algorithms such as `mod_deflate` or `mod_gzip`. Compressing files will result in a substantial performance boost and is always recommended. Two utilities that are often used are:

► `mod_deflate`

This utility is available from the following Web page:

http://httpd.apache.org/docs/2.0/mod/mod_deflate.html

► `mod_gzip`

This utility is available from the following Web page:

<http://sourceforge.net/projects/mod-gzip>

As an example, using `mod_deflate` compression can reduce a 1 MB file down to 134 KB. The result is over a 7× reduction in size. An example of enabling these compression modules can be found in the 4.3.2, “General performance tips” on page 230.

4.3.2 General performance tips

A vast array of performance optimization tips exist for boosting Web site performance. The following list details some general performance tips that you can refer to when you would like to build a high-performance store based on the WebSphere Commerce Web 2.0 Store solution.

Development tips

During development, keep the following tips in mind:

- ▶ Reduce the number of HTML tags on your pages. This will decrease download time and allow Dojo to parse your page quicker.
- ▶ Discourage using too many separate JavaScript files on a single page because they load sequentially.
- ▶ Put most of your JavaScript in .js files. Do not put inline JavaScript in a JSP unless you must. If the JavaScript does not alter the rendering of HTML, put it as close to the bottom of the page as possible so it does not interrupt the display speed of the page.
- ▶ Monitor all AJAX calls (as Debug) in hopes of minimizing the number of requests to the server.
- ▶ Optimize Dojo widget initialization. See Figure 4-17. This allows Dojo to parse your page faster.

Listing #1. Optimize widget initialization

To turn off Dojo parser, insert this code at the top of the page:

```
<script> djConfig = { parseWidgets: false, searchIds: [] }; </script>
```

Tell Dojo which element to begin parsing. Caution, Dojo will parse all nested objects contained in the "startParsingHere" <div> tag. Parsing stops when you specify parseWidgets=false

```
<div dojoType="wc:ProductQuickView" id="startParsingHere">
  <div dojoType="wc:BaseContent">
    ..
  </div>
  <div catalogEntryId = "39363" dojoType="wc:ToolTipContent">

    <div parseWidgets="false">... all other code</div>

  </div>
</div>
<script>djConfig.searchIds.push("startParsingHere");</script>
```

Figure 4-17 Optimize widget initialization

Build and deployment tips

During build and deployment, run CSSTidy and Dojo ShrinkSafe to optimize and clean your cascading style sheets and JavaScript files.

Web server configuration tips

All configurations can be enabled and modified by editing the http.conf file in your Web server.

Perform the following steps to enable and modify configurations:

1. Enable Web server compression by using code similar to that shown in Figure 4-18.

Listing #2. Turn on Web Server Compression. Use this as an example.

```
SetOutputFilter DEFLATE
BrowserMatch ^Mozilla/4 gzip-only-text/html
BrowserMatch ^Mozilla/4\.0[678] no-gzip
BrowserMatch \bMSIE\s6 no-gzip
BrowserMatch \bMSIE\s7 !no-gzip !gzip-only-text/html
SetEnvIfNoCase Request_URI \
    \.(?:gif|jpe?g|png|pdf|swf|ipk)$ no-gzip dont-vary
Header append Vary User-Agent env=!dont-vary
```

Figure 4-18 Turn on Web server compression

2. Set file timeout for static assets to a reasonable amount. An example can be seen in Figure 4-19.

Listing #3. Expire all static files after X number of days. Use this as an example.

```
ExpiresActive on
ExpiresDefault "access plus 15 minutes"
ExpiresByType image/gif "access plus 1 days"
ExpiresByType image/png "access plus 1 days"
ExpiresByType image/jpeg "access plus 1 days"
ExpiresByType image/x-icon "access plus 1 days"
ExpiresByType application/x-javascript "access plus 15 minutes"
ExpiresByType application/x-shockwave-flash "access plus 1 days"
ExpiresByType text/css "access plus 1 days"
```

Figure 4-19 Expire all static files after X number of days

3. Reduce the amount of disk access on your Web server memory cache. An example is shown in Figure 4-20. Optimize the settings based on your particular hardware.

Listing #4. Tune Your memory Cache. Use this as an example.

```
<IfModule mod_cache.c>
  CacheMaxExpire 172800
  CacheIgnoreCacheControl On
  CacheEnable mem /static
</IfModule>

<IfModule mod_mem_cache.c>
  MCacheMaxObjectCount      10000
  MCacheMaxObjectSize       50000
  MCacheMinObjectSize       0
  MCacheSize                 56000
  MCacheRemovalAlgorithm    GDSF
</IfModule>
```

Figure 4-20 Tune the memory cache

4. Ensure KeepAlive is set to true in your Web server. This is the default setting for Apache because it makes a substantial difference in performance. Ensure that your network routers are not over throttling the number of client connections to your site.

4.3.3 Advanced UI performance tips

Making pages load snappy can be broken up into a science. The goal is to have visuals display as soon as possible, which gives the impression of a quicker page load. Keep in mind that pages load sequentially. For example, Figure 4-21 on page 233 shows a page structure to be rendered.

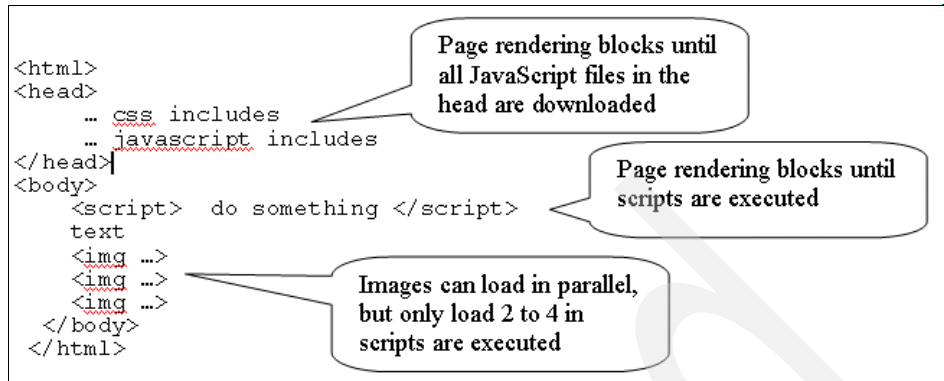


Figure 4-21 A sample page

What does this mean? Move all your `<script>` includes out of the `<head>` area if they are not needed at that level. Reduce the number of inline script sections in your code, or move them to the bottom of the page. This allows the top of the page to display to the shopper quicker.

Also, Web browsers seem to limit the number of simultaneous threads per host name. If you could increase the number of parallel downloads, download speed is improved. For example, if all your images reference the same domain as the rest of your content, you will have four parallel downloads. However, if you set up multiple sub-domains (such as: `images1.domain.com`, `images2.domain.com`, and `images3.domain.com`), and you split all the files among the different domain names, you trick the Web browsers into downloading more files in parallel. Further details about this approach can be found at the following Web page:

<http://www.websiteoptimization.com/speed/tweak/parallel>

Another technique that may make coding your JSPs easier and perform better is to have a thin layer combining all JavaScript files into a single download. More information can be found at the following Web page:

<http://rakaz.nl/extra/code/combine>

Pre-compressing all your JavaScript files and style sheets may give you slightly better performance. However, it is more difficult to configure. If you want to pre-compress and use `mod_deflate`, ensure you compress all your files and use `mod_rewrite` tags to redirect supporting browsers to the gzip content. Some rewrite rule examples have Apache check the file system to determine if each static file is compressed. You do not want to do this on a high-performing Web server. Pre-compression is not always needed if Web server caching is enabled (as mentioned in 4.2, "Debugging your Web 2.0 Store" on page 210) and large enough to hold all your JavaScript assets. If using `mod_gzip`, pre-compression functionality is supported automatically.

4.3.4 Efficiently handling requests on client side

The typical AJAX-enabled features include asynchronous request processing between client side and server side, and the multiple requests issued at the same time. This section explains how to issue and manage multiple requests and responses of the same page.

Issuing requests in parallel or serially

In the Web 2.0 Store, because of the use of asynchronous requests, the store page running in the shopper's Web browser is more interactive, responds quickly to inputs, and sections of pages can be reloaded individually. Shoppers may perceive the shopper flow to be faster or more responsive. Due to this reason, there are usually multiple requests sent to the server side in the same page. To ensure the Web 2.0 Store page behaves correctly (with a quick response in multiple requests that interact with the server side), it is important to carefully design when to serialize these requests in the same page and when these requests can be executed in parallel. Although AJAX supports issuing multiple requests at the same time, if the requests compete against the same object, the page needs to serialize the requests to avoid contention.

If the requests are for querying or getting the objects from server side for display purposes, you can issue the parallel requests to the server side. Another scenario where you can send the requests in parallel is when each request is trying to update to a different object and while multiple requests are trying to update data on the server side simultaneously. For example, reading catalog data when browsing the catalog or product information usually involves multiple requests to the server side to retrieve the corresponding catalog information displayed to the shopper. This is the query operation so it should not cause any harm when you issue these parallel requests to load the product information.

When the shopper is trying to add the item(s) to the shopping cart, the involved requests are actually updating the same business object current order. If these requests are executed in parallel, they are competing to update the same order. Your Web 2.0 Store should understand this and control these types of requests so they execute in a serial manner to avoid contention on the server side.

Co-ordination in multiple requests/responses

The previous section describes asynchronous page refreshing and how to implement it in the WebSphere Commerce Web 2.0 Store solution. Considering the case where there are multiple requests sent to the server side to update different business objects in one page, the order these response returned from the server side and received by the client side is uncertain. For example, in the product information page, the shopper can click a button to add the current item to the shopping cart. At the same time, the shopper can click another button to

add the item to the wishlist. These two requests are sent to the server serially when the shopper clicks the two buttons. Nothing can ensure which response is received by the client side first. The code in the client side of the Web 2.0 Store page should correctly identify what request to which this response corresponds and issue a corresponding get request to display the data in the refresh area. In the example, when the client side is notified that the add to wishlist response of is received, it should match it with the request adding item to wishlist, and then issue the get request to a get up-to-date wishlist that is displayed in the corresponding refresh area.

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 Web server. Point your Web browser at:

<ftp://www.redbooks.ibm.com/redbooks/SG247647>

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247647.

Using the Web material

The additional Web material that accompanies this book includes the following files:

<i>File name</i>	<i>Description</i>
TopCategoriesDisplay.jsp	Top Level JSP code
scrollablepane.js	Code for the Scrollable widget

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.

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

For information about ordering these publications, see “How to get Redbooks” on page 241. Note that some of the documents referenced here may be available in softcopy only.

- ▶ *Building Dynamic Ajax Applications Using WebSphere Feature Pack for Web 2.0*, SG24-7635
- ▶ *Planning and Managing the Deployment of WebSphere Commerce*, SG24-7588

Online resources

These Web sites are also relevant as further information sources:

- ▶ The Book of Dojo
<http://manual.dojotoolkit.org/WikiHome/DojoDotBook/Book0>
- ▶ Accessible Rich Internet Applications Wiki
<http://esw.w3.org/topic/PF/ARIA>
- ▶ ARIA Widgets and Structures
<http://esw.w3.org/topic/PF/ARIA/BestPractices/SimpleExamples>
- ▶ AJAX:WAI ARIA Live Regions
http://developer.mozilla.org/en/docs/AJAX:WAI_ARIA_Live_Regions
- ▶ Live Regions
<http://esw.w3.org/topic/PF/ARIA/BestPractices/LiveRegion>
- ▶ WAI-ARIA Live Regions
<http://juicystudio.com/article/wai-aria-live-regions.php>

- ▶ Dojo toolkit site
<http://www.dojotoolkit.org>
- ▶ Dojo API documentation
<http://api.dojotoolkit.org>
- ▶ Dojo Base 1.2.3 download
<http://dojotoolkit.org/downloads>
- ▶ Firebug lite
<http://www.getfirebug.com/lite.html>
- ▶ Dojo keys properties
<http://api.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.keys>
- ▶ dojo.dnd 1.2 technical documentation
http://docs.google.com/View?docid=d764479_11fcs7s397
- ▶ Dojo books
<http://dojotoolkit.org/book/dojo-book-0-9/part-3-programmatic-dijit-and-dojo/back-button/browser-compatibility>
- ▶ Dojo.query
<http://api-staging.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.query>
- ▶ W3C css3-selectors
<http://www.w3.org/TR/css3-selectors>
- ▶ dojo.query: A CSS Query Engine For Dojo
<http://dojotoolkit.org/node/336>
- ▶ Status Code Definitions
<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>
- ▶ Header Field Definitions
<http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html>
- ▶ Dojo Deferred
<http://redesign.dojotoolkit.org/jsdoc/dojo/HEAD/dojo.Deferred>
- ▶ CSSTidy
<http://csstidy.sourceforge.net/>
- ▶ Shrink Safe
<http://dojotoolkit.org/docs/shrinksafe>

- ▶ GZIP home page
<http://www.gzip.org/>
- ▶ Optimize Parallel Downloads to Minimize Object Overhead
<http://www.websiteoptimization.com/speed/tweak/parallel>
- ▶ Minimize HTTP Requests
<http://www.websiteoptimization.com/speed/tweak/http>
- ▶ WebSphere Commerce AJAX framework
<https://jtcid.hursley.ibm.com/commerce/index.jsp?topic=/com.ibm.commerce.madisons-starterstore.doc/refs/rsmmadisonajaxinteractions.htm>
- ▶ Best Screen Resolution to Design Web sites
<http://www.hobo-web.co.uk/tips/25.htm>
- ▶ Page Weight Tested
<http://www.marketingexperiments.com/improving-website-conversion/page-weight.html>
- ▶ Creating a Custom Distribution
<http://manual.dojotoolkit.org/book/dojo-book-0-4/part-6-customizing-dojo-builds-better-performance/dojo-build-system/creating-cust>
- ▶ Speed up your web pages with YSlow
<http://developer.yahoo.com/yslow>
- ▶ Apache Module mod_deflate
http://httpd.apache.org/docs/2.0/mod/mod_deflate.html
- ▶ mod_gzip is an Internet Content Acceleration module
<http://sourceforge.net/projects/mod-gzip>
- ▶ Combine
<http://rakaz.nl/extra/code/combine>

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks, at this Web site:

ibm.com/redbooks

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Index

A

- accessibility 24–27, 40
- account information 197
- accuracy 200
- action forms 31
- action servlet 31
- actionIds 132
- activeDrag 167
- addChild 85, 91
- adjustLeftScroll 164
- adjustRightScroll 164
- age information 38
- Ajax 35, 201
 - as part of Web 2.0 4
 - call 128
 - conversion rates 18
 - explained 4
 - invocation pattern 35–36
 - serialization 5
 - struts-actions 126
- AjaxAction 126
- AjaxAddOrderItem 127
- AjaxComponentServiceAction 13
- AjaxInterestItemAdd request 213
- AjaxOrderChangeServiceItemAdd 219
- AjaxQuickCartDisplay 219
- Ajax-style request 34, 108
- alert messages 101
- API (application programming interface)
 - accessibility 25
- APP
 - see Atom Publishing Protocol
- appendItems 164
- application errors 204
- ARIA (Accessibility Rich Internet Applications) 25–26, 40
- aroundNode 142
- ASP (active server pages) 50, 71
- asynchronous requests 234
- AT (Assistive Technology) 24
- Atom
 - as part of Web 2.0 4
- Atom Publishing Protocol

- defined 5
- Atom Syndication Format
 - defined 5
- autoScroll parameter 160
- autoScroller 164

B

- bad convention 229
- base
 - Dojo toolkit 50
- Baseline Report and Time Series Data 206
- bindDomNode 91
- black-box testing technique 200
- bookmarking 69
- BreadCrumbTrailDisplay 103
- BreadCrumbTrailDisplay widget 103
- buildscripts directory 227
- Business Object Document (BOD) 33
- Button widget 82

C

- cache 226
- cache invalidation 206
- cancelCloseOnTimeOut 155
- cascading stylesheets 229
- catalog 207
- catalog options 39
- category page 197
- CatEntrySearchListDataBean 48
- Change Flow 38, 41
- checkAcceptance 136
- checkout options 39
- class declaration 171
- className 171
- clickSelect 167
- client engine
 - defined 8
- client façade 34
- client side context information 108
- close 142
- close link 146
- closeOnTimeOut 154

- code modules 103
- code snippet 133
- CommonContextsDeclarations.js 109
- compare result 199
- compare zone
 - conversion rates 18
- connectId 142
- ConsumerDirect store 135, 183
- ContentPane widget 160
- contextMenuForWindow 91
- controller 106
 - wc.render.RefreshController 106
- controller servlet 31
- controllerId 106
 - String 106
- conversion rates
 - improving 18
- copyonly 137
- copyState 136
- core
 - Dojo toolkit 51
- Cosmetic style properties 97
- CPU 207
- CPU usage 206
- cross-browser communication 50
- CSS 50, 94, 97
 - accessibility 24
 - definitions 79
 - Web customization issues 11
- CSSTidy 228–229
- currency selection 38
- currencyCode 167
- customization
 - simplifying 12

D

- data interchange format 4
- database logs 206
- database node 206
- dataBeans 118
- db2 snapshots 206
- db2diag.log 206
- DBclean 206
- deadlock report 205
- debugging 210
- decimalPoints 167
- declareContext method 109
- declareservice tag 106

- defaultEnd 166
- defaultStart 166
- defects 213
- delayHidePopup 152
- deleteOrderItems 15
- deleteSelectedNodes 136
- department page 197
- destroy 136
- Dialog widget 86
- dialog widget 155
- dijit 79, 95
 - accessibility 26
 - defined 51
- dijit.byId 57
- dijit.Dialog 144
- dijit.form.Button 82
- dijit.form.DropDownButton 158
- dijit.MenuItem 159
- dijit.showTooltip 142
- dijit.TooltipDialog 87
- disabled 92
- disk I/O 206
- displayStatus variable 155
- DIV
 - Web customization issues 11
- div tag 82
- djConfig 53
- djConfig parameters 54
- DnD 9
 - RIA functionality 9
- DnD widget 88
- dndType 137, 140
- Document Object Model (DOM)
 - and Ajax 5
- Dojo 95
 - explained 6
- Dojo Ajax 93, 105
- Dojo API 134
- Dojo core library 52
- Dojo debug utility 28
- Dojo dialog 144
- Dojo functionality 94
- Dojo package 227
- Dojo profile 50
- Dojo toolkit 49
 - as part of Web 2.0 4
- Dojo tooltip 141
- Dojo widget initialization 230
- Dojo widgets 94

- accessibility 26
- dojo.addOnLoad 59
- dojo.back 69
- dojo.back.addToHistory 66
- dojo.back.setInitialState 66
- dojo.byId 57
- dojo.connect 60
- dojo.connect function 61
- dojo.dnd.Source 65
- dojo.dnd.Target 65, 138
- dojo.every 57
- dojo.filter 57
- dojo.forEach 57
- dojo.fromJson 59
- dojo.indexOf 57
- dojo.isArray 60
- dojo.isFunction 60
- dojo.isObject 60
- dojo.js 51, 61
- dojo.lastIndexOf 57
- dojo.map 57
- dojo.provide 52
- dojo.publish 60
- dojo.query 60, 70
- dojo.require 53, 56, 80
- dojo.require function 53
- dojo.some 57
- dojo.string.substitute 60
- dojo.subscribe 60
- dojo.toJson 59
- dojo.trim 60
- dojo.xhr 78
 - parameters 71
- dojo.xhrDelete 77
- dojo.xhrGet 60, 76
- dojo.xhrPost 60, 76
- dojo.xhrPost API 133
- dojo.xhrPut 77
- dojoType 80
- Dojox 95
- DOM
 - Dojo query engine 69
- DOM manipulation API 105, 134
- DOM Nodes 142
- DOM nodes 142
- drag and drop (DnD) 65, 135
- dragsourcetype 136
- drop area 202
- drop down list 197

- dropdown content 156
- dropdown menus 103
- DropDownButton widget 83
- dummy render context 130

E

- e-marketing spot 46, 104, 197, 199
- emotional response 200
- Enterprise Service Bus architecture
 - and SOA 6
- error 73
- error message 202
- error messages 101
- errorMessage 113
- errors 213
- errorViewName 133
- errorViewName parameters 133
- e-spot 208
- ETL (extraction, transformation, and loading) 206
- event API 93
- exception 202
- execution time 206

F

- failureHandler 112
- Fast Finder 11, 37, 42, 45–49, 165, 197, 199
 - conversion rates 19
- file content structure 93
- Firebug 213, 224, 226
- Firebug lite 51
- FirebugLite 54
- firstHandleTitle 167
- footer 103
- formErrorHandleClient 142
- fragment identifier 66
- functional scenarios 196
- functional testing 196

G

- gender information 38
- getChildren 85, 91
- getDisplayStatus 155
- getParent 92
- graphical user interface (GUI) 25
- green plus indicator 202
- gzip 233

H

- hallway usability testing 203
- handleAs 56
- hasChildren 85
- header 103
- heap fragmentation 206
- hide 144, 155
- HorizontalSlider widget 81
- horizontalSliderValue element 82
- HTML
 - accessibility 24
- HTML code 215
- HTML DIV tags 25
- HTML element 128
- HTML tag 128, 230
- HTML template 79

I

- iconClass 83, 92
- IFrame communication 51
- images 79, 94
- include 95
- incrementValue 167
- initial page request 211
- InitProperties 107
- inner tags 89
- innerHTML 106, 142
- InterestItemAdd controller command 125
- invocation pattern 36
- ioArgs 73
- isDebug 56
- isHorizontal 162
- isScrollable 162

J

- Java Client API 33
- Java Servlet 71
- JavaScript 28, 79, 95
 - accessibility 25
 - and Ajax 5
- JavaScript Object Notation (JSON) 34, 50, 105, 117, 126
 - and Atom feeds 5
 - as part of Web 2.0 4
 - data 105
 - object 106, 134
- JavaScript service object 126
- jsld 137–138

- JSP (Java Server Pages) 12, 27, 34, 50, 71, 99
 - handles 133
 - snippets 27
- JVM (Java Virtual Machine)
 - logs 206

K

- KeepAlive 232
- keyPressed 136

L

- label 83, 142
- language selection 38
- left sidebar 165
- live regions 27
- loadImage 164
- locale 56

M

- Madison's
 - capabilities 11
 - introduction 10–11
- Madisons 10
- Madisons Starter Store 38, 93, 225
- Markup parser 51
- max error rate 205–206
- memory leak 206
- Menu 89
- Menu widget 89
- MenuItem widget 89
- MerchandisingAssociations 160
- microcosm concurrent test 204
- mini shopping cart
 - streamlining 20
- MiniCartContentsJSON 133
- MiniCartContentsJSON view 133
- MiniShopCartDropDown 156
- MiniShoppingCart drop zone 140
- mobile devices 50
- mod_deflate 229
- mod_gzip 229
- mod_rewrite tags 233
- modelChanged 129
- modelChangedHandler 107
- modelChangedHandler function 130
- modelChangeedHandler 110
- modular code 98

- mousePress 155
- multi-cultural environment 203
- multilingual 203
- MVC (Model View Controller)
 - and RIAs 8
 - design pattern 30–33, 35–36
- MyAccount 95
- MyAccount Center Content 121
- MyAccount page 104
- myModule.js 52

O

- OAGIS (Open Application Group Integration Specifications) 33
- objectId
 - String 107
- onblur 63
- onClick 63, 83, 86, 92
- onClose 91
- onDndDrop 136
- onDndStart 135
- onDropDownClick 159
- onFocus 63
- onKey 155, 158
- onKeyDown 63
- onKeyPressed 63
- onMouseClicked 156
- onMouseover 145
- onOpen 91
- ontology 26
- open 142
- open source framework 31
- OpenAjax 51
- optimization 223
- optimizing response time 228
- order confirmation page 196
- order options 39
- order status tracking 38
- OrderChangeServiceItemAdd service 133
- OrderItem 126
- orderstatus 103
- outermost tag 89
- out-of-control errors 202

P

- page layout 93
- page layout structure 102, 104
- paper prototypes 203

- parseOnLoad 56
- performance 200
- Perl CGI 71
- personal information management page options 38
- PFWG (Protocols and Formats working group) 25, 40
- PHP 50, 71
- populatePopUp 148
- popupDelay 91
- popupMenuItem 89
- PopupMenuItem widget 89
- postCreate 172
- postRefreshHandler 108, 110
- priceranger 104
- product compare pag 197
- product detail page 197
- ProductAddtoCart 154
- ProductQuickInfo 144
- programmatic error 229
- promotional e-mail 38

Q

- Quick Checkout profile feature 22
- Quick Info panel 45
- quickcart 219

R

- RangeSlider 165
- reading catalog data 234
- recall 200
- red plus indicator 202
- Redbooks Web site 241
- refresh area triggers 128
- RefreshArea 27, 106, 121
- RefreshArea widget 106–107, 114
- RefreshController 101, 106
 - logic 106, 118
- RefreshJSP snippet 118
- register a user 196
- registered refresh controller 106
- registration operation 196
- registration page options 38
- relatedSource element 155
- removeChild 85, 91
- render context 108
- renderContext 107
- renderContext declaration 101
- renderContext object 118, 130

- renderContextChanged 118
- renderContextChangedHandler 106–107
- renderContextChangeHandler 110
- rePosition 155
- response time 206–207
- REST (Representational State Transfer)
 - as part of Web 2.0 4
 - defined 6
- RIA (Rich Internet Application) 10, 25
 - accessibility 24–27
 - advantages 8
 - defined 8
- right sidebar 104
- RSS feeds 19

S

- screen resolution 201
- scripted instructions 203
- scroll bar 201
- scrollable pane 199
- Scrollable widget 46
- ScrollablePane widget 160
- scrollVertical 163
- secondHandleTitle 168
- service 126
- service declaration 101, 128
- service invocation model 4
- service response 34
- serviceId 114
- service-oriented architecture (SOA)
 - and Web 2.0 6
- serviceResponse 113
- ServicesDeclaration.js 115
- setDataStore 164
- setDisabled 92
- setLabel 85
- shipping 198
- shopper 203, 234
- shopper types
 - guest 208
 - new registered shopper 208
 - registered shopper 208
- shopping cart 203, 234
 - abandonment 202
- shopping cart page 21
- shopping experience 18
- shoppingarea 95
- ShoppingCart 127

- show 144, 154
- showDelay 142
- showPopup 141, 147–148
- showThumbs 164
- showTooltip 167
- showTooltipAllTime 167
- ShrinkSafe 228–229
- SilkPerformer 205–206
- single payment method 198
- single servlet 32
- SitePen 50
- size 206
- sizeProperty 162
- SliderHorizontal widget 81
- smoothScroll 162
- snippets 95
- SOAP (Simple Object Access Protocol)
 - compared to SOA 6
- social computing 10
- software usability testing 203
- starting time 206
- states and properties 26
- store home page 196
- store structure 93
- StoreAccordion.js 229
- Struts 31, 33
- Struts Action Servlet 33
- Struts configuration XML file 125
- Struts Web application 32
- successful messages 101
- successHandler 112
- successHandler function 128
- successTest 113
- Sun Microsystems 50
- system errors 204

T

- tabular form 198
- taglibraris 99
- tamper data 224
- targetNodeIds 91
- TCOI (total cost of implementation) 95
- templateString property 172
- test machines 206
- testForChangedRC 106, 108
- thumbHeight 162
- thumbNode 164
- thumbWidth 162

- time slice 204
- timeOut milliseconds 154
- tooltip 141
- tooltip of the product quick view 197
- Tooltip widget 80
- TooltipDialog widget 86
- topcategories 103
- TopCategories menu 158–159
- TopCategoriesDisplay.jsp 99
- totalDisplayNodes 162
- tree control 25
- tuning parameters 206

U

- UltraEdit 227
- unBindDomNode 91
- updateContext method 118
- updateContextURL 108
- updater 118
- usability testing 200
- userarea 95
- userinterface 148
- UserRegistration page 141

V

- validateParameters 113
- validation error 196

W

- WAI (Web Accessibility Initiative) 25, 40
- wc sub packages 95
- wc.render.declareContext 108
- wc.render.declareRefreshController (initProperties) 107
- wc.render.getContextById 109
- wc.render.getRefreshControllerById 108
- wc.render.updateContext 109
- wc.service.declare 112
- wc.service.getServiceById 114
- wc.service.invoke() 130
- wc.widget.WCDropDownButton 159
- WcDialog 153
- WcDropDownButton 158
- wcf
 - getData tag 35
- Web 1.0 Store 200, 210–211
- Web 2.0

- advantages 7
- Web 2.0 Ajax programming framework 183
- Web 2.0 interactions 210
- Web 2.0 Store 27, 37, 93, 196, 210
 - checkout process 22
- Web developers 224
- Web performance 228
- webapp package structure 94
- weblet 207
- WebSphere Commerce Ajax APIs 95
- WebSphere Commerce Ajax framework 93, 105, 128, 134
- WebSphere Commerce Component Services 105
- WebSphere Commerce controller command 105, 125
- WebSphere Commerce runtime programming model 133
- WebSphere Commerce Web 2.0 Store 134
- widget 79
- window resize 201
- wishlist 42, 95
 - conversion rates 19
- workstation resolution 201
- WSDL (Web Service Definition Language)
 - compared to SOA 6

X

- XHR 50, 73
- xhrDelete 77
- xhrGet 75
- xhrPost 76
- xhrPost API 134
- xhrPut 77
- XML
 - as part of Web 2.0 4
- XMLHttpRequest 66
 - retrieving data 5
- XMLHttpRequest (XHR) 70

Y

- YSlow 228



WebSphere Commerce Best Practices in Web 2.0 Store

(0.5" spine)
0.475" <-> 0.875"
250 <-> 459 pages



WebSphere Commerce Best Practices in Web 2.0 Store

Understanding the technologies behind Web 2.0

This IBM Redbooks publication positions Web 2.0 Store and demonstrates how Web 2.0 is applied to an e-commerce site. This book discusses the Web 2.0 Store and Social solutions. In addition, you will gain an understanding of the contributions Dojo and Ajax apply to the Web 2.0 Starter Store.

Applying Web 2.0 to an e-commerce site

This book emphasizes some of the Web 2.0 Store solution's benefits for the client and describes the Web 2.0 features of the new Web 2.0 Store solution of WebSphere Commerce. This book discusses the Model View Controller (MVC) design pattern for Web 2.0 and provides guidance to developers who are building and customizing Ajax-based applications.

Implementing a Web 2.0 Store

This book explains how to implement your own Web 2.0 Store using Ajax and Dojo and on the basis of the Madisons store. Developers can take advantage of building their own Web 2.0 Store in more efficient ways, for example, developers quickly design, develop, test, analyze, and deploy high-quality Web 2.0 Stores.

This book describes the best practice when you test and debug the functionalities of your Web 2.0 Store, and also presents the Web 2.0 Store optimization best practice.

This book is targeted for the following audience, WebSphere Commerce Developers, WebSphere Commerce Architects, and Technical Sales Specialist.

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