Building Composite Applications

Learn about composite applications and component intercommunication

Develop components using Rational Application Developer

Assemble and deploy managed applications

Juan R. Rodriguez
Alex Barbosa Coqueiro
Belen Gonzalez Agudo
Sunil Patel
Ricardo Rossi
Rafael Sanchez
Robert Schneider
Guillermo Villavicencio
Art Whorley
Michael Zink

ibm.com/redbooks
Building Composite Applications

July 2007
**Note:** Before using this information and the product it supports, read the information in “Notices” on page xi.
# Contents

Notices ......................................................... xi
Trademarks ...................................................... xii

Preface .......................................................... xiii
The team that wrote this book ................................. xiii
Become a published author .................................... xvi
Comments welcome ............................................ xvi

Chapter 1. Introduction to SOA ................................. 1
  1.1 Service-oriented architecture - a business view ............ 2
    1.1.1 What is a “service” in service-oriented architecture? 3
    1.1.2 The SOA Foundation .................................... 4
    1.1.3 SOA life cycle .......................................... 5
    1.1.4 SOA reference architecture model ....................... 9
    1.1.5 Supporting elements of the SOA reference architecture. 12
  1.2 Service-oriented architecture - an IT view .................. 13
    1.2.1 Definition of a service-oriented architecture .......... 13
    1.2.2 SOA terms .............................................. 14
    1.2.3 Drivers for SOA ........................................ 16
    1.2.4 What is a service? ....................................... 16
  1.3 SOA governance ............................................. 18
    1.3.1 What is SOA governance? ............................... 19
    1.3.2 SOA governance in practice ........................... 20
    1.3.3 Aspects of SOA governance ............................ 21
  1.4 SOA summary ............................................... 28

Chapter 2. Composite applications ............................. 31
  2.1 Overview .................................................. 32
    2.1.1 Cooperative components ............................... 35
  2.2 Server based and managed composite applications .......... 35
  2.3 Features of a composite application ........................ 37
  2.4 Integration with WebSphere Portal .......................... 38
    2.4.1 Overview .............................................. 38
    2.4.2 Client Composite Application Infrastructure .......... 40
    2.4.3 A portal administered client .......................... 42
  2.5 Composite applications in Lotus Expeditor .................. 43
    2.5.1 Composable components in Lotus Expeditor ............. 43
    2.5.2 Component layout procedures .......................... 43
    2.5.3 Intercommunication between components ................. 45
### Chapter 7. Developing RCP Components

7.3 Creating the SendMessage portlet project ........................................ 156
7.4 Creating the ReceiveMessage portlet project ................................... 163
7.5 Creating the DemoApp composite application ................................. 171
7.6 Running the composite application .................................................. 179
7.7 Wiring portlet components. .............................................................. 182
  7.7.1 Creating WSDL files ................................................................. 183
  7.7.2 Updating the portlet.xml file ..................................................... 188
  7.7.3 Adding the property broker and defining wires ......................... 189
  7.7.4 Updating the portlet code ......................................................... 193
7.8 Running and testing the composite application ............................... 194
7.9 Troubleshooting. .............................................................................. 195

### Chapter 8. An overview of RCP components

8.1 SWT overview ................................................................................. 198
  8.1.1 SWT and Swing ......................................................................... 199
  8.1.2 Display ...................................................................................... 200
  8.1.3 Widgets .................................................................................... 208
  8.1.4 Events ...................................................................................... 209
  8.1.5 Composites and shells ............................................................... 210
  8.1.6 Layouts .................................................................................... 215
  8.1.7 Dialogs ..................................................................................... 219
  8.1.8 Resources ................................................................................ 220
  8.1.9 SWT forms ............................................................................... 226
8.2 JFace user interface framework ....................................................... 221
  8.2.1 Viewers ................................................................................... 223
  8.2.2 Dialogs .................................................................................... 226
  8.2.3 Wizards ................................................................................... 233
  8.2.4 Preferences ............................................................................. 243
  8.2.5 Field editors ............................................................................ 246

### Chapter 9. Developing RCP components

9.1 Sample scenario ............................................................................... 252
9.2 Importing existing components ....................................................... 253
9.3 Creating the SWT component ......................................................... 254
9.4 The composite application ............................................................. 261
  9.4.1 Creating the composite application project ............................... 261
  9.4.2 Laying out the components ....................................................... 261
  9.4.3 Defining the composite application launch options .................... 262
9.5 Running and testing the composite application .............................. 262
9.6 RCP component intercommunication ............................................. 264
  9.6.1 Adding component intercommunication support ....................... 264
  9.6.2 Adding wires to the composite application ............................... 277
9.7 Running the composite application in Lotus Expeditor .................. 280
Chapter 10. Integrating AWT or Swing components in composite applications
10.1 AWT and Swing applications. 284
10.2 SWT versus AWT 285
10.3 AWT components in Lotus Expeditor 285
   10.3.1 IBM Device Runtime Environment (DRE) 287
   10.3.2 The SWT-AWT bridge 287
   10.3.3 SWT-AWT bridge intercommunication 288
10.4 Sample scenario 289
   10.4.1 Installing the IBM Device Runtime Environment 289
   10.4.2 The DemoApp application 291
   10.4.3 The AWT Message component 294
   10.4.4 The composite application 305
   10.4.5 Configuring the test environment for running the application 307
   10.4.6 Summary 307

Chapter 11. Debugging the wired composite applications in Lotus Expeditor
11.1 Working with the OSGi console 312
11.2 Property Broker commands 314

Chapter 12. Integrating OLE documents and ActiveX Controls in composite applications
12.1 Overview 324
12.2 Steps to integrate an OLE object as a component of a composite application 324
12.3 Sample scenario 327

Chapter 13. Using the OSGi Event Admin
13.1 Overview 334
13.2 ITSO Car Rental scenario 334
   13.2.1 Using the OSGi Event Admin 335

Chapter 14. Managed composite applications
14.1 Overview 348
14.2 Prerequisites for declaring composite applications 349
14.3 Access control and roles 350
14.4 Provisioning 352
14.5 Definition and placement 353
   14.5.1 Portlet instances versus portlet clones 354
14.6 Providing components in a portal sense 355
   14.6.1 Placing Eclipse based components 355
   14.6.2 Placing portlets as components 360
   14.6.3 Placing a WSRP portlet viewer 361
19.1 Architectural overview ................................................................. 478
19.2 The database component ............................................................. 480
   19.2.1 Customer model ................................................................. 480
   19.2.2 Car model ................................................................. 482
   19.2.3 Contract model ................................................................. 483
   19.2.4 Database helper functions ..................................................... 484
19.3 The booking component ............................................................... 485
19.4 The car selection component ......................................................... 488
19.5 The credit card component ........................................................... 493
19.6 The customer contract list component ............................................. 494
19.7 The weather component .............................................................. 497
   19.7.1 Tasks for producer portals ...................................................... 498

Chapter 20. Assembling Portal-administered composite applications . 501
20.1 Overview .............................................................................. 502
20.2 Composite Application Infrastructure ........................................... 502
20.3 Network Client Installer .............................................................. 503
   20.3.1 Portlets installed by NCI ......................................................... 503
   20.3.2 Templates and applications ..................................................... 507
20.4 Sample scenario ....................................................................... 508
   20.4.1 Preparing the application for deployment ................................. 509
   20.4.2 Setting up the application in Portal ......................................... 514
   20.4.3 Assembling the application ..................................................... 516
   20.4.4 Deploying the composite application ....................................... 532

Chapter 21. Component development tools ........................................... 537
21.1 Lotus Domino Designer 8 .............................................................. 538
   21.1.1 The user interface ................................................................. 539
   21.1.2 Composite application design elements ..................................... 540
   21.1.3 Property Broker Editor ......................................................... 541
   21.1.4 WSDL file handling ............................................................. 542
21.2 Lotus Component Designer ........................................................... 544

Chapter 22. NSF component development ......................................... 547
22.1 Composite applications in Lotus Notes 8 ....................................... 548
   22.1.1 Terminology ...................................................................... 548
   22.1.2 Architecture overview ........................................................ 549
   22.1.3 Components in a composite application ..................................... 549
   22.1.4 Lotus Script APIs for inter-component communication .............. 551
22.2 Composable components in Lotus Notes ........................................ 554
22.3 Developing NSF components ....................................................... 557
   22.3.1 Introduction ...................................................................... 557
   22.3.2 Software requirements ........................................................ 560
   22.3.3 Existing notes application ..................................................... 560
Chapter 26. Provisioning the Lotus Expeditor clients with Device Management

26.1 Overview ................................................................. 680
26.2 OSGi overview ...................................................... 680
26.3 OMA DM overview ............................................... 681
26.4 Lotus Expeditor Client management ......................... 681
26.5 Architecture ......................................................... 681
  26.5.1 Device Manager ................................................. 682
  26.5.2 Enterprise Management Agent ............................. 684
  26.5.3 Device Manager database ................................. 684
  26.5.4 Device Manager Console ................................. 684
26.6 Lotus Expeditor Client sample scenario ..................... 685
  26.6.1 Enrolling the Lotus Expeditor Client in Device Manager .............................. 685
  26.6.2 Software distribution for Lotus Expeditor Client ........................................ 688
  26.6.3 Software removal for Lotus Expeditor Client ............................................. 697
  26.6.4 Software update for Eclipse features .......................................................... 701
  26.6.5 Client-side troubleshooting ................................. 701
26.7 Troubleshooting with Device Manager ....................... 702

Appendix A. Developing Notes applications .................. 705
Notes discussion forum .............................................. 706
  Software required ...................................................... 706
  Creating a new discussion database ........................... 706
  Adding additional features to the discussion database .... 709

Appendix B. Additional material ................................... 721
Locating the Web material .......................................... 721
Using the Web material .............................................. 721
  System requirements for downloading the Web material .................................. 722
  How to use the Web material ........................................ 722

Related publications ............................................... 723
IBM Redbooks ............................................................. 723
Other publications ..................................................... 723
Online resources ........................................................ 723
How to get IBM Redbooks ......................................... 724
Help from IBM ............................................................. 725

Index .......................................................................... 727
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

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

<table>
<thead>
<tr>
<th>AIX®</th>
<th>Lotus Notes®</th>
<th>Sametime®</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloudscape™</td>
<td>Lotus®</td>
<td>Tivoli®</td>
</tr>
<tr>
<td>DB2®</td>
<td>Notes®</td>
<td>VisualAge®</td>
</tr>
<tr>
<td>developerWorks®</td>
<td>Rational®</td>
<td>WebSphere®</td>
</tr>
<tr>
<td>Domino®</td>
<td>Redbooks®</td>
<td>Workplace™</td>
</tr>
<tr>
<td>IBM®</td>
<td>Redbooks (logo)</td>
<td>Workplace Managed Client™</td>
</tr>
</tbody>
</table>

The following terms are trademarks of other companies:

SAP, and SAP logos are trademarks or registered trademarks of SAP AG in Germany and in several other countries.

Oracle, JD Edwards, PeopleSoft, Siebel, and TopLink are registered trademarks of Oracle Corporation and/or its affiliates.

EJB, Java, Java 2D, JavaServer, JavaServer Pages, JDBC, JDK, JNI, JRE, JSP, J2EE, J2SE, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

ActiveX, Excel, Internet Explorer, Microsoft, MSDN, Visual Basic, Windows, Win32, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

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

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

This IBM® Redbooks® publication covers composite applications for desktop solutions that require multiple types of applications to run in a cooperative environment by providing intercommunication between components. Service-oriented architecture (SOA) composite applications deliver high levels of business services and this book covers the architecture, available tools, component considerations, as well as assembling, deploying and wiring components in WebSphere® Portal and Rich Client Platforms, such as Lotus® Expeditor and Lotus Notes® 8.

Lotus Expeditor is a client platform for end-to-end smart client application solutions. Expeditor provides services to install and manage these applications and allows users to easily launch and switch among these applications. Expeditor leverages Eclipse technology to provide a client solution that runs on multiple operating systems. The benefits of composite applications include reducing the total cost of ownership for client solutions through reusability of existing components and the sharing of these components across applications, migrating existing applications to run on the client platform, controlling access to applications based on roles assigned to users and improving the user experience by allowing composite application components to easily communicate between them.

This publication is targeted at architects, developers and administrators who need to understand the implemented architecture to develop, assemble and deploy composite applications in Lotus Expeditor V6.1.

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.

Juan R. Rodriguez is a Consulting IT Professional and Project Leader at the IBM ITSO Center, Raleigh. He has a master's degree in Computer Science from Iowa State University. He writes extensively and teaches IBM classes worldwide about Web technologies and information security. Before joining the IBM ITSO, he worked at the IBM laboratory in Research Triangle Park, North Carolina, as a designer and developer of networking products.
Alex Barbosa Coqueiro is a Certified IT Specialist in the Lotus Software Group in IBM Brazil. He holds a master's degree in Software Engineering. His areas of expertise include WebSphere Portal, WebSphere Portlet Factory, Workplace™ Dashboard Framework, and Lotus Expeditor. He is a member of IBM University Ambassadors and teaches Java™ architecture at Faculdade de Informática e Administração Paulista (FIAP) College in Sao Paulo, Brazil.

Belen Gonzalez Agudo is an IT Specialist in IBM Software Group, Spain. She has eight years of experience in the e-business field. Her areas of expertise include J2EE™ application development and portal solutions. She has co-authored several IBM Redbooks related to portlet application development and portal tools. She holds a degree in Computer Science Engineering from Universidad Autonoma de Madrid, Spain.

Sunil Patel is an Advisory Software Engineer in IBM Software Group. He works with Business Partners, customers, and universities and he provides technical enablement via workshops, articles, technical support and other services. His expertise primarily includes IBM WebSphere Portal and Lotus Collaboration technologies. He holds a master’s degree in Information Technology from the University of North Texas and he is completing a Ph.D. in IT Change Management.

Ricardo Rossi is an Advisory Software Engineer at IBM Innovation Center in Dallas, Texas. He is responsible for developing and teaching Lotus Expeditor and Notes 8 workshops as well as participating in proof of concept and other customer engagements. He holds a master’s degree in Information Technology and Management from the University of Texas at Dallas.

Rafael Sanchez is an IT Architect with IBM Spain. He has bachelor’s and master’s degrees in Computer Engineering, a bachelor’s degree of Business Administration, and a Music Teacher degree from the Royal Conservatory of Music of Madrid. He has many years of experience in Pervasive Computing, including mobile telephony, embedded devices, wireless networks, and RFID. He is currently involved with Computer and Telecommunication Engineering Ph.D. studies on artificial intelligence and biologically inspired computing.
Robert Schneider works as an IT Specialist for the Lotus Technical Sales department in IBM Germany. He is a Customer Advocate for IBM Workplace and Lotus Collaboration technologies. He has 20 years of experience in the computer business arena. In recent years, Robert has focused on the new, Java-based client platforms, such as Workplace Managed Client™, Expeditor, and the upcoming Hannover technology.

Guillermo Villavicencio holds a degree in Informatics Engineering from the Pontifical Catholic University of Peru. He is a Research and Development Manager for AVATAR SAC, an IBM Premier Business Partner. He has been the architect for several e-business projects including wireless and portal solutions. His current areas of expertise are centered around Web technologies and SOA. He writes and teaches classes worldwide on Web technologies and Java application development using Eclipse platforms.

Art Whorley is a Certified Consulting IT Specialist in the WPLC Software organization specializing in technical sales support for Enterprise Access and Client Technology solutions. Art holds a bachelor’s degree in Electrical Engineering Technology. He is a member of the Advisory Board for the Enterprise Access Competency Council and is actively involved in mentoring other IBM sales and technical teams as well as Business Partners.

Michael Zink is a Consulting IT Specialist with IBM Austria. Currently he is the Technical Sales Leader for Enterprise Access and Client Technologies for North East Europe. Michael has worked in various positions for IBM. His current focus is designing and building composite applications for Lotus Expeditor and Lotus Notes, as well as server managed clients for taking enterprise applications beyond the data center.

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

Bob Balfe, Craig Wolpert, Jo Grant
IBM Westford, Massachusetts

Janet Dmitrovich, Charles Imperato, Rahul S. Kurane, Keith Carter
IBM Austin, Texas
Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbooks publication dealing with specific products or solutions, while getting hands-on experience with leading-edge technologies. You'll 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 Redbooks to be as helpful as possible. Send us your comments about this or other 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 email 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
Chapter 1. Introduction to SOA

Service-oriented architecture (SOA) is an approach to defining integration architectures based on the concept of a service. Applications collaborate by invoking each other's services, and services can be composed into larger sequences to implement business processes.

This chapter introduces service-oriented architecture from a business perspective and from an IT perspective.
1.1 Service-oriented architecture - a business view

The primary goal of service-oriented architecture (SOA) is to align the business world with the world of information technology (IT) in a way that makes both more effective. SOA is about the business results that can be achieved from having better alignment between the business and IT.

SOA starts from the premise that all businesses have a business design. A business design describes how that business works – the processes that it performs; the organizational structure of the people and finances within that business; the business’ near-term and long-term goals and objectives; the economic and market influences that affect how that business achieves its goals; the rules and policies that condition how the business operates.

Most businesses have a written form of their high level business plan – the high level definition that states the business’ purpose. Few businesses, however, have a written form of their business design. Many of those who have documented their business design have trouble keeping their design up to date with what they actually practice. Business processes evolve as businesses respond to shifts in the marketplace, regulations, or product innovations; this evolution usually happens without reflecting those changes in the formal design of the business.

However, even if the business design has not been documented, or even if what is documented is now obsolete, there is nonetheless a business design in effect. A fundamental premise of SOA is that if the business design can be transcribed and maintained there is a potential for leveraging that captured design information.

Even if the business design is not used to communicate between the business and IT organizations, it can nonetheless be a valuable tool to help businesses understand what they are doing and how. Beyond that, however, the business design becomes an essential tool in communicating requirements between the business and the IT organization. The business can identify those elements of the design that should be automated and what within that design should be performed by people, creating a blueprint of the information systems that are created to support that automation.

By deriving the information system design from the business design you can more easily drive changes into the information system at the rate and pace of change in the business design. Furthermore, the information system can be used as a catalyst for change in the business design. It is from this correspondence that SOA delivers on the promise of more flexible businesses through more flexible IT.
This correspondence is represented as the SOA Lifecycle, in which the business process is modeled, assembled, deployed and monitored in an iterative manner. This transforms the information system from being one of merely a “cost of doing business” to a fundamental tool for enabling a business to be more competitive, profitable and responsive to changes in the marketplace.

To achieve this synergism between the business and IT domains we need to employ a number of capabilities:

- A formalism and language for capturing the business design
- A methodology for translating the business design into a set of information system artefacts to implement that design
- An infrastructure for hosting those implementation artefacts that is as flexible to changes in its marketplace as the business itself needs to be
- A place for retaining the correlation between the business design and the information system that can be used to identify and fix failures in executing on the goals and constraints of the business design
- A means by which we can manage the system to ensure those goals are met.

These capabilities improve the flow of the business process through SOA Lifecycle iterations.

### 1.1.1 What is a “service” in service-oriented architecture?

We refer to the practice of deriving an information system design from a business design as service-oriented architecture. The business process and the tasks from which it is composed can be collectively thought of roughly as services. Thus, the business design is essentially a composition of services and the policies and conditions that must be applied to their use which form the information system design.

However, there remains the question of “what is a service?” Is it a function within an application? Are all application functions services? Does SOA include system services? Coming up with a single, mathematically precise definition that applies universally to all situations can be hugely complicated. In practice, such precision is not necessary to achieving value and success from SOA.

An underlying premise in the application of SOA to information technology is the principle of loose coupling – that is, avoiding or at least encapsulating temporal, technological and organizational constraints in the information system design. This same principle applies also to the definition of service – the rules used to define services in one context may not be applicable in another. What is important is that whatever definition we arrive at, it should originate from the primary concerns and constraints of that context. As a generalization, a service is
a **repeatable task within a business process**. So, if you can identify your business processes, and within that, the set of tasks that you perform within the process, then you can claim that the tasks are services and the business process is a **composition of services**.

However, note that certain tasks can be decomposed into business processes in their own right. The order-entry process includes, among other things, a task to confirm availability of the items being ordered. The confirm-availability task is itself a business process that includes, for example, the tasks of checking the on-hand inventory, verifying the supply pipeline, and possibly creating a background request and determining its availability. Thus, business processes are themselves services; there is a principle of recursive decomposition implied in the term service. If taken far enough we could easily claim that everything is a service. This, obviously, is not useful – at some point treating everything as a service would yield an incredibly inefficient over-generalization of the problem space. You should exercise this principle of recursive decomposition only to the extent that you legitimately need flexibility within your business design.

From this definition of service, service-orientation is a way of integrating your business as a set of linked services. If you can define the services in each of your vertical and horizontal lines-of-business, you can begin to link those LOBs by composing their services into larger business processes. Likewise, you can decompose the main services of your LOBs into a set of more basic services that can then be easily recomposed either to change LOB processes, or to interlink your LOBs at a lower level of their capabilities. Similarly, you can use the same principles of composition to create links with your business partners to both automate those relationships and to gain more efficiency from them. One consequence of service orientation is flexibility: you gain the ability to iteratively optimize your business design, unhampered by inflexible IT structures.

A **service-oriented architecture**, then, is an **architectural style for creating an Enterprise IT Architecture that exploits the principles of service-orientation** to achieve a tighter relationship between the business and the information systems that support the business.

### 1.1.2 The SOA Foundation

What should be clear is that SOA is not just about technology. IBM views SOA as a holistic relationship between the business and the IT organization. SOA encompasses the tools and methodologies for capturing business design, and using that design information to help improve the business. It also encompasses the tools, programming model and techniques for implementing the business design in information systems. It encompasses the middleware infrastructure for hosting that implementation. SOA encompasses the management of that implementation to ensure availability to the business, and to ensure efficient use
of resources in the execution of that implementation. It encompasses the establishment of who has authority and the processes that are used to control changes in the business design and its implementation in the information system. And ultimately, SOA accelerates the time-to-value for these benefits.

The SOA Foundation is a comprehensive architecture and set of offerings, technologies, and practices that address all of these things about SOA. To avoid the connotation that SOA is only about technology we deliberately choose not to use the term SOA “Platform”.

1.1.3 SOA life cycle

The SOA life cycle (see Figure 1-1) begins with modeling your business (capturing your business design) including the key performance indicators of your business goals and objectives, translating that model into an information system design, deploying that information system, managing that deployment, and using the results coming out of that environment to identify ways to refine the business design. It is a premise of the lifecycle that feedback is cycled to and from phases in iterative steps of refinement and that the model may actually be built using reverse-engineering techniques or other means to facilitate the needs of the business.

Figure 1-1 SOA life cycle
The lifecycle is then layered on a backdrop of a set of governance processes that ensure that compliance and operational policies are enforced, and that change occurs in a controlled fashion and with appropriate authority as envisioned by the business design.

**Model**
Modeling is the process of capturing your business design from an understanding of business requirements and objectives and translating that into a specification of business processes, goals and assumptions – creating an encoded model of your business.

Capturing your business design using a rigorous approach offers the potential to gain better insight into your business, by, for example using tools to reason about the design and its supporting rationale. In particular, we can use the model to simulate how your business processes will actually run. A sophisticated modeling approach lets you perform “what-if” scenarios that reflect your understanding of the actual number of process instances, contacts, quantities, incoming traffic, etc. that you may experience in your business. The process can then be simulated using those parameters to predict the effect that process will have on your business and on your IT systems. If you do not achieve the hoped-for results then you can change your process definition to try to improve your results. You can go on to refine your processes as you have modeled them to optimize your business performance even before ever investing in an implementation of those processes.

Your model will also capture key performance indicators – business metrics that are important measurements of your business. This could include, for example, a measure of the new accounts that you have opened in a given month. These key performance indicators are input to the assembly of your application and later, when the application is in production, collected and reported back to you. You will be able to use that information to determine how well your business is performing. You can use the correlation between your business design in your actual implementation in the information system to determine whether bottlenecks in your performance are due to limitations in your business design or limitations in the information system that automates your design.

For more information, go to the following Web site:


**Assemble**
You can use your business design to communicate with the IT organization – to assemble the information system artefacts that will implement the business design. The enterprise architect working with the business analyst can begin to convert the business design into a set of business process definitions and
activities deriving the required services from the activity definitions. They can work with the software architect to flesh out the design of the services.

During the process of resolving a design and implementation of your modeled business processes and services, you should search your existing asset inventories – your existing programs – to find application components that already meet your needs. Some application components will fit perfectly; some will have to be re-factored; and some will have to be augmented to meet the requirements of the design. These existing assets should be rendered as services for assembly into composite applications.

It is also possible that some of the your legacy is so heavily bound into a tight relationship with presentation and data logic and other business functions that you simply cannot extract any re-usable componentry from those programs. In these cases you will have to decide whether and how to rewrite these functions as new services, and how to migrate the processes that depend on those old programs.

Any new services required by the business design will have to be created. Software developers should use the SOA programming model to create these new services.

Final assembly includes applying the set of policies and conditions to control how your applications operate in your production environment. This might include, for example, business and government regulations, but can also include critical operational characteristics such as packaging, localization constraints, resource dependency, integrity control, and access protection.

For more information, go to the following Web sites:


**Deploy**

The deploy phase of the lifecycle includes a combination of creating the hosting environment for your applications and the actual deployment of those applications. This includes resolving the application’s resource dependencies, operational conditions, capacity requirements, and integrity and access constraints.

A number of concerns are relevant to construction of the hosting environment – including the presence of the already existing hosting infrastructure supporting legacy applications and pre-existing services. Beyond that, you need to consider
appropriate platform offerings for hosting your user interaction logic, business process flows, business-services, access services, and information logic.

You need to consider the techniques you will employ for ensuring availability, reliability, integrity, efficiency, and service ability.

For more information, go to the following Web sites:


Manage

Turning now to the manage phase of the lifecycle, you need to consider how to maintain the operational environment and the policies expressed in the assembly of the SOA applications deployed to that environment. This includes monitoring performance of service requests and timeliness of service responses; maintaining problem logs to detect failures in various system components; detecting and localizing those failures; routing work around them; recovering work affected by those failures; correcting problems; and restoring the operational state of the system.

The manage phase also includes managing the business model – tuning the operational environment to meet the business objectives expressed in the business design, and measuring success or failure to meet those objectives. SOA is distinguished from other styles of enterprise architecture by its correlation between the business design and the software that implements that design, and its use of policy to express the operational requirements of the business services and processes that codify the business design. The manage phase of the lifecycle is directly responsible for ensuring those policies are being enforced, and for relating issues with that enforcement back to the business design.

Managing the system also involves performing routine maintenance, administering and securing applications, resources and users, and predicting future capacity growth to ensure that resources are available when the demands of the business call for it.

For more information, go to the following Web sites:


Lifecycle flow
Progression through the lifecycle is not entirely linear. In fact, changes to key performance information in the Model phase often need to be fed directly in to the Management phase to update the operational environment. Constraints in the Deploy phase, such as limiting assumptions about where resources are located in the system, may condition some of the Assembly phase decisions. And, occasionally, information technology constraints established in the Assembly phase will limit the business design created during the Model phase – for example, the cost of wide-area wireless communication with remote handheld devices may be prohibitive to deploying a field force to rural locations and therefore needs to be reflected back into the business design.

1.1.4 SOA reference architecture model

The SOA reference architecture model attempts to decompose the functional underpinnings of your application design. Notice white space between architecture elements in Figure 1-2. This is intended to imply our emphasis on maintaining a clean separation of concerns. The separation enables us to focus attention on the special skills that are required for each section of the overall architecture – enabling you to optimize your resources to the skills required for a given topic. This specialization avoids the situation that everyone on the team needs to understand everything about the entire system to be effective at anything they do in part of it. This should lower the cost of training, enable more efficient implementations, and enable the construction of tools optimized for specific skill sets.

The reference architecture model attempts to be comprehensive – spanning all of the requirements of the SOA Foundation. The parts in light-green in Figure 1-2 are the parts in which you will deploy application software to capture the domain logic specific to your business design. These are the boxes labeled Interaction Services, Process Services, Information Services, Partner Services, Business Application Services and Access Services.

The other parts of the architecture exist to assist the rest of the SOA lifecycle. You use these other parts in the modeling of your business design, construction and assembly of your software, deployment of your applications, and management of your operational system and the business design you have...
implemented. You may even customize these other parts with metadata or software you write to control or optimize those environments, but generally not with logic that is specific to your business design.

Interaction Services
Interaction services are about the presentation logic of the business design – components that support the interaction between applications and end-users. Also we recognize that interactions with the external world are not limited to just interactions with humans. In some cases, interaction logic orchestrates the interface to industrial robots, vehicles, sensors, RFID devices, environmental control systems, process control equipment, etc.

Process Services
Process services include various forms of compositional logic – the most notable of which are business process flows and business state machines (finite-state machines for business composition). We consider both kinds of composition mechanisms, plus other forms such as business rules and decision tree processing, as well as more ad-hoc forms of composition, to be equally valid approaches to choreographing service composition.

For more information about this topic, go to the following Web sites:
Business Application Services
Business application services implement your core business logic. These are service components created specifically as services within a business model and that represent the basic building blocks of your business design – services that are not decomposable within the business model, but that can be composed to form higher level services.

Information Services
Information services contain the data logic of your business design. This logic exists at two levels. On the surface, information services provide access to the persistent data of your business. This can include query statements for retrieving the information you care about or referential integrity checks on the information manipulated by these services. These data services are available to the business application as services – often constructed with specific domain model semantics so that they appear for all intents and purposes as business application services.

For more information, go to the following Web site:

Access Services
Access services are dedicated to integrating legacy applications and functions into the service-oriented architecture. This includes simple wrapping of those functions and rendering them as services (in the case where the existing function is a good match with the semantic requirements of the business model in which it will be used), or in more complex cases augmenting the logic of the existing function to better meet the needs of the business design. In the latter case, the access service may in fact invoke multiple legacy functions to achieve the semantic requirements of the service. In other architectures we have often referred to these access services as adapters.

For more information, go to the following Web site:

Partner Services
Partner services capture the semantics of partner interoperability that have a direct representation in the business design. This can, for example, include the
policies and constraints that other businesses must conform to work with your business – including business vertical requirements such as the need to conform to specific industry message and interchange standards like EDIFACT, SWIFT, RosettaNet, etc. It can involve the business logic of managing how partners are selected, and which ones are used as a preference over others in particular circumstances.

1.1.5 Supporting elements of the SOA reference architecture

The remaining portions of the reference architecture are directly relevant to the SOA Foundation, and in fact may even be aspects that you can contribute code to or customize to meet your needs, but generally will not contain functional aspects of the business design and the application logic that goes directly to implementing that business design. In general, these areas of the functional architecture have more to do with the exploitation of information technology as a means for implementing the business design. They in effect represent information technology concerns within the overall architecture.

Business Innovation and Optimization Services

These services primarily represent the tools and the metadata structures for encoding your business design, including your business policies and objectives. Business innovation and optimization is achieved by capturing your business design and then introspecting on that design to improve it through a combination of iterative refinement and analysis of real-time business metrics.

Business innovation and optimization services exist in the architecture to help you capture, encode, analyze and iteratively refine your business design. The services also include tools to help you simulate your business design and to use those results to predict the effect that design, or changes to that design, will have on your business. In addition, these services include tools to help you define your key performance indicators – that is, those business objectives and general metrics that you want to monitor.

For more information, go to the following Web sites:


IT Service Management
Once your application has been deployed to the information system, it needs to be managed along with the IT infrastructure on which it is hosted. IT service management represents the set of management tools used to monitor your service flows, the health of the underlying system, the utilization of resources, the identification of outages and bottlenecks, the attainment of service goals, the enforcement of administrative policies, and recovery from failures.

For more information, go to the following Web sites:


Infrastructure Services
Infrastructure services form the core of the information technology environment for hosting SOA applications. It is through these services that we are able to build a reliable system to provide efficient utilization of resources, ensure the integrity of the operational environment, balance workload to meet service level objectives, isolate work to avoid interference, perform maintenance, secure access to confidential business processes and data, and simplify overall administration of the system.

1.2 Service-oriented architecture - an IT view

In the previous section, we described service-oriented architecture from a business point of view. In this section, we describe SOA from an IT point of view.

1.2.1 Definition of a service-oriented architecture

Following are a few definitions of SOA. Note the common terms used in the definitions.

Component model
Service-oriented architecture is a component model that interrelates an application’s different functional units, called services, through well-defined interfaces and contracts between these services. The interface is defined in a neutral manner that should be independent of the hardware platform, the operating system, and the programming language in which the service is implemented. This allows services, built on a variety of such systems, to interact with each other in a uniform and universal manner.
This feature of having a neutral interface definition that is not strongly tied to a particular implementation is known as loose coupling between services. The benefit of a loosely-coupled system is its agility and ability to survive evolutionary changes in the structure and implementation of the internals of each service that makes up the whole application.

**Application architecture**

Service-oriented architecture is an application architecture in which all functions, or services, are defined using a description language and have invocable interfaces that are called to perform business processes. Each interaction is independent of each and every other interaction and the interconnect protocols of the communicating devices (that is, the infrastructure components that determine the communication system do not affect the interfaces). Because interfaces are platform-independent, a client from any device using any operating system in any language can use the service.

Though built on similar principles, SOA is not the same as Web services, which indicates a collection of technologies, such as SOAP and XML. SOA is more than a set of technologies and runs independent of any specific technologies.

**Integration architecture**

Service-oriented architecture is an integration architecture approach based on the concept of a service. The business and infrastructure functions that are required to build distributed systems are provided as services that collectively, or individually, deliver application functionality to either end-user applications or other services.

SOA specifies that within any given architecture, there should be a consistent mechanism for services to communicate. That mechanism should be coupled loosely and should support the use of explicit interfaces.

### 1.2.2 SOA terms

Figure 1-3 highlights the key terms used to describe a service-oriented architecture.
A service is representative of a repeatable business task. Services are used to encapsulate the functional units of an application by providing an interface that is well defined and implementation independent. Services can be invoked (consumed) by other services or client applications.

Service orientation defines a method of integrating business applications and processes as linked services.

Service-oriented architecture (SOA) can be different things to different people depending on the persons role and context (business, architecture, implementation, operational). From a business perspective, SOA defines a set of business services composed to capture the business design that the enterprise wants to expose internally, as well as its customers and partners. From an architecture perspective, SOA is an architectural style that supports service orientation. At an implementation level, SOA is fulfilled using a standards based infrastructure, programming model and technologies such as Web services. From an operational perspective, SOA includes a set of agreements between service consumers and providers that specify the quality of service, as well as reporting on the key business and IT metrics.

A composite application is a set of related and integrated services that support a business process built on an SOA.
1.2.3 Drivers for SOA

The main driver for SOA is to define an architectural approach that assists in the flexible integration of IT systems. Organizations spend a considerable amount of time and money trying to achieve rapid, flexible integration of IT systems across all elements of the business cycle. The drivers behind this objective include:

- Increasing the speed at which businesses can implement new products and processes, can change existing ones, or can recombine them in new ways
- Reducing implementation and ownership costs of IT systems and the integration between them
- Enabling flexible pricing models by outsourcing more fine-grained elements of the business than were previously possible or by moving from fixed to variable pricing, based on transaction volumes
- Simplifying the integration work that is required by mergers and acquisitions
- Achieving better IT utilization and return on investment
- Achieving implementation of business processes at a level that is independent from the applications and platforms that are used to support the processes

SOA prescribes a set of design principles and an architectural approach to achieve this rapid flexible integration.

1.2.4 What is a service?

Having outlined SOA as being an architectural approach to defining integration architectures based on services, it is important to define what is meant by a service in this context in order to fully describe SOA and to understand what can be achieved by using it. A service can be defined as any discrete function that can be offered to an external consumer. This function can be an individual business function or a collection of functions that together form a process.

There are many additional aspects to a service that we must also consider in the definition of a service within an SOA. The most commonly agreed-on aspects are that services:

- Encapsulate reusable business functions
- Are defined by explicit, implementation-independent interfaces
- Are invoked through communication protocols that stress location transparency and interoperability
Reusable business functions

A service can be any business function. In an SOA, however, it is preferable that the function is genuinely reusable. The goal of a service in an SOA is that it can be used and reused by one or more systems that participate in the architecture. For example, while the reuse of a Java logging API can be described as *design time* (when a decision is made to reuse an available package and bind it into application code), the intention of SOA is to achieve the reuse of services at:

- **Runtime**
  Each service is deployed in one place and one place only and is invoked remotely by anything that must use it. The advantage of this approach is that changes to the service (for example, to the calculation algorithm or the reference data on which it depends) need only be applied in a single place.

- **Deployment time**
  Each service is built once but redeployed locally to each system or to the set of systems that must use it. The advantage of this approach is increased flexibility to achieve performance targets or to customize the service (perhaps according to geography).

Explicit implementation independent interfaces

The use of explicit interfaces to define and to encapsulate service function is of particular importance to making services genuinely reusable. The interface should encapsulate only those aspects of process and behavior that are used in the interaction between the service consumer and the service provider. An explicit interface definition, or contract, is used to bind a service consumer and a service provider. It should specify only the mutual behavior that is required for the interaction and should specify nothing about the implementation of the consumer or the provider.

By explicitly defining the interaction in this way, those aspects of either system (for example the platform on which they are based) that are not part of the interaction are free to change without affecting the other system. This implementation-independent interface allows either system to change implementation or identity freely.

Figure 1-4 illustrates the use of explicit interfaces to define and encapsulate services functions.
Communication protocols that stress location transparency

SOA does not specify that any specific protocol be used to provide access to a service. A key principle in SOA is that a service is not defined by the communication protocol that it uses but instead, should be defined in a protocol-independent way that allows different protocols to be used to access the same service.

Ideally, a service should only be defined once, through a service interface, and should have many implementations with different access protocols. This type of definition helps to increase the reusability of any service definition.

1.3 SOA governance

SOA is a compelling technique for developing software applications that best align with business models. However, SOA increases the level of cooperation and coordination required between business and information technology (IT), as
well as among IT departments and teams. This cooperation and coordination is provided by SOA governance, which covers the tasks and processes for specifying and managing how services and SOA applications are supported.

1.3.1 What is SOA governance?

In general, governance means establishing and enforcing how a group agrees to work together. Specifically, there are two aspects to governance:

- Establishing chains of responsibility, authority, and communication to empower people, determining who has the rights to make what decisions.
- Establishing measurement, policy, and control mechanisms to enable people to carry out their roles and responsibilities.

Governance is distinct from management in the following ways:

- Governance determines who has the authority and responsibility for making the decisions.
- Management is the process of making and implementing the decisions.

To put it another way, governance says what should be done, while management makes sure it is getting done.

A more specific form of governance is IT governance, which does the following:

- Establishes decision-making rights associated with IT.
- Establishes mechanisms and policies used to measure and control the way IT decisions are made and carried out.

That is, IT governance is about who’s responsible for what in an IT department and how the department knows those responsibilities are being performed.

SOA adds the following unique aspects to governance:

- Acts as an extension of IT governance that focuses on the lifecycle of services to ensure the business value of SOA.
- Determines who should monitor, define, and authorize changes to existing services within an enterprise.

Governance becomes more important in SOA than in general IT. In SOA, service consumers and service providers run in different processes, are developed and managed by different departments, and require a lot of coordination to work together successfully. For SOA to succeed, multiple applications need to share common services, which means they need to coordinate on making those services common and reusable. These are governance issues, and they're much
more complex than in the days of monolithic applications or even in the days of reusable code and components.

As companies use SOA to better align IT with the business, companies can ideally use SOA governance to improve overall IT governance. Employing SOA governance is key if companies are to realize the benefits of SOA. For SOA to be successful, SOA business and technical governance is not optional, it is required.

1.3.2 SOA governance in practice

In practice, SOA governance guides the development of reusable services, establishing how services will be designed and developed and how those services will change over time. It establishes agreements between the providers of services and the consumers of those services, telling the consumers what they can expect and the providers what they're obligated to provide.

SOA governance doesn't design the services, but guides how the services will be designed. It helps answer many thorny questions related to SOA: What services are available? Who can use them? How reliable are they? How long will they be supported? Can you depend on them to not change? What if you want them to change, for example, to fix a bug? Or to add a new feature? What if two consumers want the same service to work differently? Just because you decide to expose a service, does that mean you are obligated to support it forever? If you decide to consume a service, can you be confident that it will not be shut down tomorrow?

SOA governance builds on existing IT governance techniques and practices. A key aspect of IT governance when using object-oriented technologies like Java 2 Platform, Enterprise Edition (J2EE) is code reuse. Code reuse also illustrates the difficulties of IT governance. Everyone thinks reusable assets are good, but they're difficult to make work in practice: Who's going to pay to develop them? Will development teams actually strive to reuse them? Can everyone really agree on a single set of behavior for a reusable asset, or will everyone have their own customized version which isn't really being reused after all? SOA and services make these governance issues even more important and thus, their consequences even more significant.

Governance is more of a political problem than a technological or business one. Technology focuses on matching interfaces and invocation protocols. Business focuses on functionality for serving customers. Technology and business are focused on requirements. While governance gets involved in those aspects, it focuses more on ensuring that everyone is working together and that separate efforts are not contradicting each other. Governance does not determine what
the results of decisions are, but what decisions must be made and who will make them.

The two parties, the consumers and the providers, have to agree on how they're going to work together. Much of this understanding can be captured in a service-level agreement (SLA), measurable goals that a service provider agrees to meet and that a service consumer agrees to live with. This agreement is like a contract between the parties, and can, in fact, be a legal contract. At the very least, the SLA articulates what the provider must do and what the consumer can expect.

SOA governance is enacted by a center of excellence (COE), a board of knowledgeable SOA practitioners who establish and supervise policies to help ensure an enterprise's success with SOA. The COE establishes policies for identification and development of services, establishment of SLAs, management of registries, and other efforts that provide effective governance. COE members then put those policies into practice, mentoring and assisting teams with developing services and composite applications.

Once the governance COE works out the policies, technology can be used to manage those policies. Technology doesn't define an SLA, but it can be used to enforce and measure compliance. For example, technology can limit which consumers can invoke a service and when they can do so. It can warn a consumer that the service has been deprecated. It can measure the service's availability and response time.

A good place for the technology to enforce governance policies is through a combination of an enterprise service bus (ESB) and a service registry. A service can be exposed so that only certain ESBs can invoke it. Then the ESB/registry combination can control the consumers' access, monitor and meter usage, measure SLA compliance, and so on. This way, the services focus on providing the business functionality, and the ESB/registry focuses on aspects of governance.

1.3.3 Aspects of SOA governance

SOA governance is not just a single set of practices, but many sets of practices coordinated together. The sections that follow provide a brief overview of the various aspects of SOA governance.

Service definition
The most fundamental aspect of SOA governance is overseeing the creation of services. Services must be identified, their functionality described, their behavior scoped, and their interfaces designed. The governance COE may not perform these tasks, but it makes sure that the tasks are being performed. The COE
coordinates the teams that are creating and requiring services, to make sure needs are being met and to avoid duplicate effort.

Often, it is not obvious what should be a service. The function should match a set of repeatable business tasks. The service's boundaries should encapsulate a reusable, context-free capability. The interface should expose what the service does, but hide how the service is implemented and allow for the implementation to change or for alternative implementations. When services are designed from scratch, they can be designed to model the business; when they wrap existing function, it can be more difficult to create and implement a good business interface.

An interesting example of the potential difficulties in defining service boundaries is where to set transactional boundaries. A service usually runs in its own transaction, making sure that its functionality either works completely or is rolled back entirely. However, a service coordinator may want to invoke multiple services in a single transaction (ideally through a specified interaction like WS-AtomicTransactions). This task requires the service interface to expose its transaction support so that it can participate in the caller's transaction. But such exposure requires trust in the caller and can be risky for the provider. For example, the provider may lock resources to perform the service, but if the caller never finishes the transaction (it fails to commit or roll back), the provider will have difficulty cleanly releasing the resource locks. As this scenario shows, the scope of a service and who has control is sometimes no easy decision.

Service deployment lifecycle
Services do not come into being instantaneously and then exist forever. Like any software, they need to be planned, designed, implemented, deployed, maintained, and ultimately, decommissioned. The application lifecycle can be public and affect many parts of an organization, but a service's lifecycle can have even greater impact because multiple applications can depend on a single service.

The lifecycle of services becomes most evident when you consider the use of a registry. When should a new service be added to the registry? Are all services in a registry necessarily available and ready for use? Should a decommissioned service be removed from the registry?

While there is no one-size-fits-all lifecycle that is appropriate for all services and all organizations, a typical service development lifecycle has five main stages:

1. Plan
   A new service that is identified and is being designed, but has not yet been implemented or is still being implemented.
2. Test

Once implemented, a service must be tested. Some testing may need to be performed in production systems, which use the service as though it were active.

3. Active

This is the stage for a service available for use and what we typically think of as a service. It’s a service, it’s available, it really runs and really works, and it hasn’t been decommissioned yet.

4. Deprecate

This stage describes a service which is still active, but won’t be for much longer. It is a warning for consumers to stop using the service.

5. Sunset

This is the final stage of a service, one that is no longer being provided. Registries may want to keep a record of services that were once active, but are no longer available. This stage is inevitable, and yet frequently is not planned for by providers or consumers.

Sunsetting effectively turns the service version off, and the sunset date should be planned and announced ahead of time. A service should be deprecated within a suitable amount of time before it is sunsetting, to programmatically warn consumers so that they can plan accordingly. The schedule for deprecation and sunsetting should be specified in the SLA.

One stage which may appear to be missing from this list is “maintenance.” Maintenance occurs while a service is in the active state; it can move the service back into test to reconfirm proper functionality, although this can be a problem for existing users depending on an active service provider.

Maintenance occurs in services much less than you might expect; maintenance of a service often involves not changing the existing service, but producing a new service version.

Service versioning

No sooner than a service is made available, the users of those services start needing changes. Bugs need to be fixed, new functionality added, interfaces redesigned, and unneeded functionality removed. The service reflects the business, so as the business changes the service needs to change accordingly.

With existing users of the service, however, changes need to be made judiciously so as not to disrupt their successful operation. At the same time, the needs of existing users for stability cannot be allowed to impede the needs of users desiring additional functionality.
Service versioning meets these contradictory goals. It enables users satisfied with an existing service to continue using it unchanged, yet allows the service to evolve to meet the needs of users with new requirements. The current service interface and behavior is preserved as one version, while the newer service is introduced as another version. Version compatibility can enable a consumer expecting one version to invoke a different but compatible version.

While versioning helps solve these problems, it also introduces new ones, such as the need to migrate.

**Service migration**

Even with service versioning, a consumer cannot depend on a service, or more specifically, a desired version of that service, to be available and supported forever. Eventually, the provider of a service is bound to stop providing it. Version compatibility can help delay this “day of reckoning” but won’t eliminate it. Versioning does not obsolete the service development lifecycle, but it enables the lifecycle to play out over successive generations.

When a consumer starts using a service, it is creating a dependency on that service, a dependency that has to be managed. A management technique is for planned, periodic migration to newer versions of the service. This approach also enables the consumer to take advantage of additional features added to the service.

However, even in enterprises with the best governance, service providers cannot depend on consumer migration alone. For a variety of reasons, for example legacy code, manpower, budget, priorities, some consumers may not migrate in a timely fashion. Does that mean the provider must support the service version forever? Can the provider simply disable the service version one day after everyone should have already migrated?

Neither of those extremes is desirable. A good compromise is a planned deprecation and sunsetting schedule for every service version, as described in “Service deployment lifecycle” on page 22.

**Service registries**

How do service providers make their services available and known? How do service consumers locate the services they want to invoke? These are the responsibilities of a service registry. It acts as a listing of the services available and the addresses for invoking them.

The service registry also helps coordinate versions of a service. Consumers and providers can specify which version they need or have, and the registry then makes sure to only enumerate the providers of the version desired by the consumer. The registry can manage version compatibility, tracking compatibility
between versions, and enumerating the providers of a consumer's desired version or compatible versions. The registry can also support service states, like test and deprecated, and only make services with these states available to consumers that want them.

When a consumer starts using a service, a dependency on that service is created. While each consumer clearly knows which services it depends on, globally throughout an enterprise these dependencies can be difficult to detect, much less manage. Not only can a registry list services and providers, but it can also track dependencies between consumers and services. This tracking can help answer the age-old question: Who's using this service? A registry aware of dependencies can then notify consumers of changes in providers, such as when a service becoming deprecated.

**Service message model**

In a service invocation, the consumer and provider must agree on the message formats. When separate development teams are designing the two parts, they can easily have difficulty finding agreement on common message formats. Multiply that by dozens of applications using a typical service and a typical application using dozens of services, and you can see how simply negotiating message formats can become a full-time task.

A common approach for avoiding message format chaos is to use a canonical data model. A canonical data model is a common set of data formats that is independent of any one application and shared by all applications. In this way, applications don't have to agree on message formats, they can simply agree to use existing canonical data formats. A canonical data model addresses the format of the data in the message, so you still need agreement around the rest of the message format, for example header fields, what data the message payload contains, and how that data is arranged, but the canonical data model goes a long way toward reaching agreement.

A central governance board can act as a neutral party to develop a canonical data model. As part of surveying the applications and designing the services, it can also design common data formats to be used in the service invocations.

**Service monitoring**

If a service provider stops working, how will you know? Do you wait until the applications that use those services stop working and the people that use them start complaining?

A composite application, one that combines multiple services, is only as reliable as the services it depends on. Since multiple composite applications can share a service, a single service failure can affect many applications. SLAs must be defined to describe the reliability and performance consumers can depend on.
Service providers must be monitored to ensure that they're meeting their defined SLAs.

A related issue is problem determination. When a composite application stops working, why is that? It may be that the application head, the UI that the users interface with, has stopped running. But it can also be that the head is running fine, but some of the services it uses, or some of the services that those services use, are not running properly. Thus it's important to monitor not just how each application is running, but also how each service (as a collection of providers) and individual providers are also running. Correlation of events between services in a single business transaction is critical.

Such monitoring can help detect and prevent problems before they occur. It can detect load imbalances and outages, providing warning before they become critical, and can even attempt to correct problems automatically. It can measure usage over time to help predict services that are becoming more popular so that they can run with increased capacity.

**Service ownership**

When multiple composite applications use a service, who is responsible for that service? Is that person or organization responsible for all of them? One of them; if so, which one? Do others think they own the service? Welcome to the ambiguous world of service ownership.

Any shared resource is difficult to acquire and care for, whether it's a neighborhood park, a reusable Java framework, or a service provider. Yet a needed pooled resource provides value beyond any participant's cost: Think of a public road system.

Often an enterprise organizes its staff reporting structure and finances around business operations. To the extent that an SOA organizes the enterprise's IT around those same operations, the department responsible for certain operations can also be responsible for the development and run time of the IT for those operations. That department owns those services. Yet the services and composite applications in an SOA often don't follow an enterprise's strict hierarchical reporting and financial structure, creating gaps and overlap in IT responsibilities.

A related issue is user roles. Because a focus of SOA is to align IT and business, and another focus is enterprise reuse, many different people in an organization have a say in what the services will be, how they will work, and how they'll be used. These roles include business analyst, enterprise architect, software architect, software developer, and IT administrator. All of these roles have a stake in making sure the services serve the enterprise needs and work correctly.
An SOA should reflect its business. Usually this means changing the SOA to fit the business, but in cases like this, it may be necessary to change the business to match the SOA. When this is not possible, increased levels of cooperation are needed between multiple departments to share the burden of developing common services. This cooperation can be achieved by a cross-organizational standing committee that, in effect, owns the services and manages them.

**Service testing**
The service deployment lifecycle includes the test stage, during which the team confirms that a service works properly before activating it. If a service provider is tested and shown to work correctly, does the consumer need to retest it as well? Are all providers of a service tested with the same rigor? If a service changes, does it need to be retested?

SOA increases the opportunity to test functionality in isolation and increases the expectation that it works as intended. However, SOA also introduces the opportunity to retest the same functionality repeatedly by each new consumer who doesn't necessarily trust that the services it uses are consistently working properly. Meanwhile, because composite applications share services, a single buggy service can adversely affect a range of seemingly unrelated applications, magnifying the consequences of those programming mistakes.

To leverage the reuse benefits of SOA, service consumers and providers need to agree on an adequate level of testing of the providers and need to ensure that the testing is performed as agreed. Then a service consumer need only test its own functionality and its connections to the service, and can assume that the service works as advertised.

**Service security**
Should anyone be allowed to invoke any service? Should a service with a range of users enable all users to access all data? Does the data exchanged between service consumers and providers need to be protected? Does a service need to be as secure as the needs of its most paranoid users or as those of its most lackadaisical users?

Security is a difficult but necessary proposition for any application. Functionality needs to be limited to authorized users and data needs to be protected from interception. By providing more access points to functionality (that is, services), SOA has the potential to greatly increase vulnerability in composite applications.

SOA creates services that are easily reusable, even by consumers who ought not to reuse them. Even among authorized users, not all users should have access to all data the service has access to. For example, a service for accessing bank accounts should only make a particular user's accounts available, even though the code also has access to other accounts for other
users. Some consumers of a service have greater needs than other consumers of the same service for data confidentiality, integrity, and nonrepudiation.

Service invocation technologies must be able to provide all of these security capabilities. Access to services has to be controlled and limited to authorized consumers. User identity must be propagated into services and used to authorize data access. Qualities of data protection have to be represented as policies within ranges. This enables consumers to express minimal levels of protection and maximum capabilities and to be matched with appropriate providers who may, in fact, include additional protections.

For more information, go to the following Web sites:


1.4 SOA summary

Service-oriented architecture and Web services enable new opportunities for more flexible, rapid, and widespread integration in a model that is consistent with the exposure of business function as services. SOA and Web services offer the choreography of those services into processes that can be modeled, executed, and monitored with features such as:

- SOA defines concepts and general techniques for designing, encapsulating, and invoking reusable business functions through loosely bound service interactions. Most of the techniques have been proven individually in previous technologies or design styles. SOA unites them in an approach that is intended to bring encapsulation and reuse to the enterprise level.

- Web services provide an emerging set of open-standard technologies that can be combined with proven existing technologies to implement the concepts and techniques of service-oriented architecture.

- Industry support for Web services standards, interoperability among different implementations of Web services, and the infrastructure technology that is required to support a service-oriented architecture give technology customers increasingly mature and sophisticated technologies that are suitable for service-oriented architecture implementation.
These techniques and technologies give you the tools that are required to implement flexible service-oriented architectures and to evolve toward an on demand business model. However, SOA is an architectural approach, not a technology or a product. In order to implement an SOA, you must have the infrastructure to support the architecture, such as an Enterprise Service Bus and a service registry and repository.

For more information, go to the following Web site:

Composite applications

This chapter introduces the key concepts related to composite applications. You will discover how Lotus Expeditor V6.1 enables the deployment of composite applications on centrally managed clients, which can seamlessly operate in connected and disconnected environments, and can be tightly integrated with a WebSphere Portal server environment.

The topics presented in this chapter are covered in depth in separate chapters.
2.1 Overview

There are many challenges associated with application development and delivery in today’s On Demand world. Companies face requirements to grow their businesses and increase revenue; however, they must also balance these requirements against the costs of ever increasing needs to develop the company’s information technology infrastructure. Flexibility is critical for business agility and for offering new business and customer services in response to competitive pressures. Along with these challenges comes an increasing focus on employee productivity and operational efficiency. So how does an organization confront the daunting technology activities involved in integration or development of new products, markets, business services, among other areas, while maintaining existing services and leveraging existing technology resources? Service Oriented Architecture (SOA) provides a methodical approach to addressing these challenges by providing an extensible framework that allows growth, development, and agility in the information technology infrastructure.

For those not totally familiar with SOA, let us quickly review the concept of service-oriented architecture. A service is defined as a “repeatable business task”. For example, tasks that are performed repeatedly throughout any given time frame (for example, checking customer credit, pulling down new work orders, etc.) can be componentized and automated to the extent possible and feasible. Service Oriented refers to the concept of integrating business by linking these “services” together to deliver a complete solution. Considering these definitions, Service Oriented Architecture is the architectural approach for implementing the linked business services concept. In the context of SOA, Composite Applications are groups of business services programmatically linked together to deliver a comprehensive business solution.

With this understanding, it is clear that SOA provides a foundation that is specifically targeted at building business applications. The definition of SOA from Service Oriented Architecture for Dummies by Judith Hurwitz, Robin Bloor and Carol Baroudi is “an architecture for building business applications as a set of loosely coupled black-box components orchestrated to deliver a well-defined level of service by linking together business processes.”¹

Considering the definition of SOA, it is important to note that composite applications themselves are not extensions to SOA, but instead, the extensions are the components that make up the composite applications. In the Expeditor client environment, a composite application is an aggregation of components that, when wired together, provide the ability for the components to communicate

and interact with each other. These client components are the true extension of
the SOA architecture.

There are several characteristics that are essential for true SOA
implementations. First, SOA implementations should embrace open standards
such as SOAP, WSDL, XML, HTTP/HTTPS/JMS, J2EE, Eclipse, OSGi, etc, for
new application development. These standards provide maximum flexibility in
terms of application portability and protect software assets from obsolescence
associated with changes in proprietary implementations. From an application
integration perspective, SOA simplifies the integration process. An added benefit
is savings realized by creating an application once and reusing across multiple
lines of businesses (LoBs). For example, an account creation service may be the
same across multiple LoBs so the asset could be used in each LoB application.

Second, the architected implementation should provide the flexibility that
simplifies integration with former or existing applications as solution components,
where these components may include embedded browsers, ActiveX®, Web
services, Eclipse based Rich Client Platform (RCP), intelligent forms, host
access, native Windows® applications, portlets, Web containers, etc. Effectively,
integration is dependent on the ability for applications to call other applications
using existing standards as defined in the previous paragraph. There should be
virtually no limit to the possible composite application permutations.

Third, the initial driving force behind SOA has primarily been Web services;
composite applications have primarily been portal based. However, with the
introduction of Lotus Expeditor, a new dimension has been added to SOA, the
concept of extending SOA to the client. Accordingly, the principles of SOA should
also apply to the client. As a matter of fact, to be thorough the client SOA focus
should also take into account that today’s On Demand business requires a
solution architecture which incorporates the needs of mobile employees. The key
here is to develop a consistent architectural strategy for the entire enterprise
from the client to the back-end, delivering an end-to-end SOA solution.

Fourth, a well-designed SOA solution should include user considerations. The
user interface should aggregate the solution components in a manner that
reduces end user retraining by preserving the application look and feel. Where it
is practical and conducive, some functions can be moved from back-end servers
to the clients, thereby decreasing application response times, decreasing the
network load and reducing latency while offloading back-end servers. The “plug
and play” fashion of new runtimes should allow applications to be loaded and
unloaded on the fly. For example, an application could start another application
for any given need, and then stop that application when it is no longer required
as part of runtime. Composite applications embracing the OSGi standard can
establish parameters to automatically unload the stopped program thereby
minimizing the footprint and freeing up system resources. Additionally, there
should be provisions to manage dependencies with minimal user intervention (for example, App A requires App B. So if it is unavailable on the client, it will go search for it on the server). Together, these considerations translate to an improved user experience. For mobile users, additional provisions should be built in for offline operations.

Finally, to deliver on the SOA promise to “orchestrate” components and services, the ability to centrally manage and control clients should be a part of any SOA architected solution. Centralized management ensures compliance with corporate standards, enables automated installation and configuration of applications, and automates the delivery of pertinent information.

So then, how do composite application components tie into and extend SOA? In the previously stated definition, SOA specifies building loosely coupled applications and treating them as “black-boxes”. Well-designed Composite applications implement this architectural approach by providing for building business applications as described; however, they should take it a step further, providing integration of existing applications with other existing, as well as new, applications. Reusable modules of code, written with standards compliance in mind, allow the black-box approach by providing components with well-defined and documented access criteria which perform specific business processes. For example, an Order Entry class may be developed with methods to perform the following functions:

- Enable selection of various related items to be ordered
- Query suppliers for the available inventories and prices
- Consolidate the items and prices from multiple vendors into a single order
- Enforce a policy to limit orders to a set amount
- Present a screen for final review and submission to approver

The SOA concept of linking together business processes is the centerpiece of composite applications. For example, once the Order Entry class is developed, the module (or black-box) could be utilized as a component of other composite applications. The developer of the composite application would only need to utilize the methods surfaced by the Order Entry class. Then as part of the composite application scenario, a Sametime® plug-in could be developed, and linked to the Order Entry module, to enable the Order Entry application to generate an automated Sametime message to the order approver.

We can extend this scenario to account for a mobile sales force by building-in offline capabilities. For example, at the beginning of the business day, the business application could send queries to the suppliers and store the item and price information in a local database on the client device. The sales person would then process orders throughout the day without concern as to the status of
the connectivity. When offline, transactions could be queued for delivery once connectivity is reestablished. Furthermore, device management could be employed to install and configure the sales applications, and to sync price changes to the local client databases as necessary.

As you read through the information contained in the rest of this book, the above concepts will be expanded in further detail with specific focus on Lotus Expeditor. Examples will be provided to graphically reinforce the written concepts and scenarios.

### 2.1.1 Cooperative components

Whenever you are facing an application that is homogeneous, which is a sole application composed of several other different heterogeneous applications but behaves like a unique application, you are probably facing a composite application.

In order to behave like a unique application, a composite application needs to offer inter-component interaction capabilities. This means that components of a composite application are required to be able to work and interact with other components. For the remainder of this book, we will refer to these kind of components as cooperative components.

Cooperative components which make up a composite application should remain loosely-coupled. This would prevent, for example, the composite application to fail or behave incorrectly if one or more components are dynamically added or removed from it. We will unveil more composite applications features in the rest of this chapter, but first we are going to analyze the difference between server based and managed client composite applications.

### 2.2 Server based and managed composite applications

Nowadays we are quite used to Web applications made up of different fragments which are aggregated at the server side, such as portal applications. But this pattern is giving way to a new means of executing composite applications at the client side, therefore enabling the client to run a composite application in a non-connected environment.

---

**Important:** Server based and managed client composite applications are both defined, described or composed at the server side. However, server based composite applications are instantiated at the server side, while managed client composite applications are instantiated at the client side.
Server based composite application pattern

Figure 2-1 illustrates a server based composite application pattern. When the client requests a page to the Web or portal server, it is actually asking it to generate a formatted page with information from different sources.

Internally, the Web or portal server analyzes the application model to determine what components make up the composite application, applies role-based access based on the user’s identity, retrieves the corresponding information to be presented, looks up the layout, and finally aggregates all the content in a formatted page. Finally, it will send the resulting page to the client, which will display it with its Web browser.

Note: Portal applications are a specific type of application, in particular a portal application is a collection of pages, portlets, policies and roles.

A typical example of a server-based composite application infrastructure is a portal server serving formatted Web pages after Web browsers' requests at the clients. In that case, a requested page contains one or more portlets, which is accessible according to a certain policy, and is able to cooperate between them by means of a wiring mechanism.

Managed client composite application pattern

In the case of a managed client composite application, the instantiation of the composite application is not made at the server side but at the client side. In other words, an XML document of the application model is deployed to the client. This XML document describes the instructions to create the aggregation on the client of the different components composing the application. The client then looks all these components up locally and aggregates them into a graphical representation. These components usually access local information which could be received from a remote server or even another application. If it happens that a required component is not found locally, it will be retrieved remotely. This is possible because the XML document contains enough information to indicate where the required component could be downloaded from.
2.3 Features of a composite application

To be considered as a composite application, an application needs to comply with a certain architectural design, which is defined by a set of features. Even if it is not an exhaustive list, the following shows the main desirable features of a composite application:

- Homogeneity to behave as a unique application to different heterogeneous applications or components
- Flexibility to use service-oriented architecture (SOA) principles such as loosely-coupling and reusability
- Intercommunication between components
- Richer user experience to aggregate a variety of application types into a single client view
- Security such as authentication, roles and data confidentiality
- Transactional applications, which get the information from multiple sources
- Reuse computing assets
- A uniform and consistent graphical user interface
- Composition of parts or components
- Aggregates a variety of application types into a single client view
Provides anytime/anywhere access in a semi-connected environment. This is not a composite application feature but Lotus Expeditor enhances these capabilities.

2.4 Integration with WebSphere Portal

This section describes how Lotus Expeditor extends the composite application pattern from WebSphere Portal server in the following ways:

- It enables composite applications to be instantiated at the client side, emulating the same process taking place in a typical WebSphere Portal server deployment.
- Any Lotus Expeditor client can be centrally managed from a WebSphere Portal server.

2.4.1 Overview

Portal-managed composite applications can be described and deployed on WebSphere Portal server as they are centrally managed from the portal. Figure 2-3 depicts a typical portal-managed configuration.
One of the new features of this version of Lotus Expeditor that surprises newcomers is the definition of Eclipse components as portal components and the subsequent projection of the applications made up of these components into the client platform. By projecting views through WebSphere Portal server, the following benefits are achieved:

- A common aggregation and deployment model for both portal composite applications and rich client composite applications, and the use of a portal programming model at the client side.
- Composite applications can be rendered locally on the client platform.
- Composite applications can be stored in a catalog on the server.
- WebSphere Portal server policy is leveraged, because the client platform’s behavior is based on roles and access control definitions on the server side, thus providing centrally administered role-based access control capabilities.
- WebSphere Portal server administrators can dynamically modify composite applications running in client platforms.
- Portlet applications can be deployed to client platforms and then be executed either while connected or in a local and disconnected way.
In order to deploy and execute composite applications at the client side of Lotus Expeditor V6.1, it now integrates a Composite Application Infrastructure (CAI). Moreover, in order to allow a WebSphere Portal server instance to centrally manage Lotus Expeditor clients, an add-on component named Network Client Installation (NCI) needs to be installed on it. Both additions are described in the following subsections.

### 2.4.2 Client Composite Application Infrastructure

The CAI, at the client platform, is the composite application runtime environment, which is necessary to install and execute certain composite applications, in particular those applications composed in a WebSphere Portal server environment.

To instantiate a composite application on the Lotus Expeditor client, for example, the client platform needs to know what are the components of that composite application, where to find those components, and how those components interact with each other. All this information, required in order to instantiate a composite application on the client side, is encoded in an XML stream. Thus we can see the CAI composite application provisioning as XML-driven. This XML stream acts as a template and contains the necessary information to instantiate a composite application, such as the list of components which make up the application, the layout of these components, the links between components, the roles and access permissions to prevent unauthorized access to applications and components, and any other information required for the proper instantiation of this application.

The XML stream describing a composite application can be obtained from a static definition deployed to the local file system, a definition contained within a plug-in, or might also be retrieved from a portal server in a portal-managed environment.

The CAI embedded at the client platform provides several services, the following provides a description of some of them:

- Application programming interfaces (APIs) that mimic WebSphere portal Server Composite Application Infrastructure:
  - Security with managed settings: a framework that manages portal policies defined at the WebSphere Portal server for composite applications. The portal policy settings defined for a certain resource in WebSphere Portal server are likewise applied to this resource on the client platform. Internally, the framework retrieves policy and preference settings and value pairs defined on a back-end system, such as WebSphere Portal server, and stores them in the Eclipse preferences store implemented on the client platform. Policies are kept up-to-date by means of scheduled
synchronization jobs. Moreover, the client platform implements a federated policy architecture in which policy providers identify a precedence order for policy retrieval.

**Note:** A portal policy is a collection of settings that influences the behavior of a portal resource and the experience that users have when working with it. Using a policy, it is possible to limit or extend the functionality provided by a resource based on the role and associated permissions defined for a given user.

– Synchronization: it is important to note that Lotus Expeditor not only extends a composite application paradigm to rich client devices, but it also enhances it because Lotus Expeditor allows the execution of composite applications even in a non-connected environment. This is because of the synchronization services being embedded in the client platform.

Composite applications, whose components implement the synchronization interface, are able to synchronize their data so that the application can be available for use while the client is non-connected, that is when the client has either no network or when a connection to the WebSphere Portal server is not available. Whenever a user of a composite application desires to have any of the composing components available in a non-connected mode, it must register the application for synchronization. Nonetheless, the code, which is required for a component to be able to synchronize its data, must be provided by the component itself, using SyncML or another synchronization service registered with the Synchronization Manager.

– Property broker: this functionality, available with WebSphere Portal server, has been projected to the client platform.

**Note:** In WebSphere Portal server, the property broker element allows two portlets to interact through the use of a Property and Action registry and a wiring mechanism.

This enables the client platform to support Property to Action model communication. More information about this Lotus Expeditor feature can be found in 2.5.3, “Intercommunication between components” on page 45, and in Chapter 5, “Component communication” on page 85, which presents a more in-depth discussion on this topic.

► Application definition: as stated before, the composite applications layout, a list of components, policies, and additional inherent information is encoded in an XML stream. The CAI needs to be able to process composite application XML description files and extract the XML fragments called domain objects,
which describe each of the components within a composite application. With the obtained domain object information, the CAI reconstructs the composite application by deserializing each of the required components.

- **Serialization service**: we have just seen that the CAI needs to make domain objects, which have been serialized at the server side and synchronized to the client platform, available in order to reconstruct the composite application. The CAI’s serialization service allows the client platform to do this, by reading XML fragments and deserializing the corresponding serialized component.

### 2.4.3 A portal administered client

Lotus Expeditor V6.1 client platform can be considered a portal administered client because it can install, load and run applications defined by a WebSphere Portal server. To do so, we require Lotus Expeditor’s Network Client Installer (NCI) component.

NCI installs a set of portlets and portlet enhancements that allow the Lotus Expeditor client to be managed from a portal server. Listed below are the additional components installed at the portal server:

- **Workbench Administration portlet**
- **Downloads portlet**

  This portlet allows a user to initiate the installation of the Lotus Expeditor client platform from the WebSphere Portal server.

- **Rich Client Layout Administration portlet**

  This portlet enhances WebSphere Portal server as it allows the application composer to specify options and properties for pages and page components of rich client composite applications. It is integrated in the WebSphere Portal server’s Page Customizer as a new tab titled *rich client*. Recall that the Page Customizer contains portlets for editing the layout, content, and appearance of pages.

- **Policy Type Editor for Eclipse Preferences**

  An extension to the Resource Policies portlet through which Eclipse preferences for portal administered clients can be managed.

- **Rich Client Managed Browser Administration portlet**

  A placeholder to place a Web browser as a portlet, and lay it out on a portal page. Used by the administrator it allows you to aggregate local or remote Web applications within a portal-defined perspective.

- **Rich Client Web Services Remote Portlets (WSRP) Viewer enablement portlet**
- Rich Client View Placeholder portlet
  Acts as another placeholder for an Eclipse view and can be placed on a page. This portlet can be cloned and placed on a page by using the WebSphere Portal server's Page Customizer, in order to aggregate an Eclipse view into a composite application.

**Note:** In addition, WSRP enables publishing JSR168 compliant portlets to be accessed (consumed) by Lotus Expeditor client side.

## 2.5 Composite applications in Lotus Expeditor

Lotus Expeditor V6.1 does much more than just enabling the creation of composite applications. It allows the integration of different types of applications, thus forming a heterogeneous environment, in a consistent way. In other words, Lotus Expeditor V6.1 has brought the portal integration model to the rich client.

### 2.5.1 Composable components in Lotus Expeditor

With Lotus Expeditor you can build a composite application from any component employing views based on many different technologies such as:

- **Abstract Window Toolkit (AWT)**
- **Embedded Browser**
  Web application based user interfaces are displayed within an embedded browser view that is part of a predefined perspective provided by the platform. The embedded browser is a configurable and manageable browser that you can embed in a client application.
- **JSR168 standard portlet viewer**
- **Standard Widget Toolkit (SWT) components**
- **Web containers such as servlets and JSPs**
- **WSRP viewer to locally display a portlet which is actually running on a remote portal server**

### 2.5.2 Component layout procedures

You should not forget that a composite application is a visual application made of visual components. These components are required to be laid out in order to be presented to the final user. Lotus Expeditor offers two different ways to lay out components of a composite application:

- **Declarative layout**
Programmatic layout

Before providing more information about each of these models for application layout, it is important to point out that both models can be mixed in the same rich client.

Declarative layout

In order to use the declarative layout, we must be working in a portal-managed environment, because we are required to reuse the concepts and tools for composite application composition, offered by WebSphere Portal server. The process required to declare the layout of a Lotus Expeditor composite application, in this case, is closely related to the process for WebSphere Portal server composite applications. That is, components are all seen as portlets and they all must be placed on top of a common substrate or page.

Because Lotus Expeditor makes composite applications of more components than just portlets, we must wrap all those non-portlet components as proxy portlets. Remember that in order to perform this kind of functionality, you must install Lotus Expeditor NCI to extend WebSphere Portal server capabilities to manage Lotus Expeditor clients. Another key element required is the CAI, because as explained before, it handles the instantiation of composite applications described as XML streams.

It is also important to note that, what a composite application composer sees when laying out a certain application might not be the same as what the final user gets when accessing the application. It depends on the roles of the final user and its access rights. In a portal-managed environment, security features on WebSphere Portal server allow the declaration of roles and access control definitions in order to control access to applications and application components. Furthermore, the pieces of code of all non-visible components for a certain user is not deployed to the Lotus Expeditor Client.

More information about how to declare a composite application layout by means of WebSphere Portal server and Lotus Expeditor extensions to WebSphere Portal server can be found in Chapter 16, “Assembling and deploying managed client composite applications” on page 399.

Programmatic layout

Programmatic layout must be regarded as an alternative whenever declarative layout is not possible. This is true, for example, when we are using a non-portal-managed environment.

The most straightforward way to programmatically layout a Lotus Expeditor Composite Application is to put its components into a perspective. In Eclipse technology, a perspective defines a set of visual components, namely editors and
views, arranged in an initial layout for a particular role or task. In order to lay the components in a perspective, Eclipse programming model’s extension point based definition mechanism needs to be used.

More information about programmatically laying out composite applications components can be found in 14.9, “Laying out applications programmatically” on page 367.

2.5.3 Intercommunication between components

In Lotus Expeditor V6.1, the property broker enables the communication between components in a Composite Application. This kind of communication allows different components to interact in a loosely-coupled way.

Description of the property broker mechanism

The property broker available on the client platform allows for declarative properties, actions and wires to be used among completely decoupled components. In particular, the property broker detects changed properties and invokes the corresponding actions, which are connected or wired, as defined by the wires that are registered. This mechanism, might just look like a typical publish and subscribe communication, but in addition the property broker mechanism is driven by declarative markup because all the details on properties, actions and wires is defined in XML format.

The cooperative component that provides a property is called the source component, while the properties that the source component publishes are named output properties. On the other hand, the cooperative component that receives a property is called the target component, and the properties that are received by the target component are called input properties.

The property broker allows actions to be defined for the target component, which is triggered whenever any received property matches a related input property. See Figure 2-4.
In order to allow an output property from a source component to be received at a target component as an input property, we must interconnect both components with what is known as a \textit{wire}.

\begin{quote}
\textbf{Note:} The concept of \textit{wiring} comes from IBM WebSphere Portal server, and has been integrated into Lotus Expeditor. In WebSphere Portal, wiring is a technique for connecting cooperative portlets so that changes in the source portlet automatically update the target portlets without any user interaction.
\end{quote}

Whenever creating a wire, you must remember that both properties, that is the output property on the source target and the input property on the target component, must have the same data type. Figure 2-5 shows wiring components with the property broker.
Communication schemes available

Not all the components of Lotus Expeditor Composite Application can communicate between one another. There are four types of communication schemes available:

- Portlet to portlet
  
  This is an inter-portlet communication between two JSR168 portlets. When an action is invoked on a JSR168 portlet, the Portlet Container determines the action, and fires a propertyChanged event through the property broker. In turn, the property broker determines the wire and invokes the processAction on the corresponding target JSR168 portlet.

- Portlet to Rich Client Platform (RCP) component
  
  When communicating information from a portlet to an RCP component, which is between a JSR168 portlet and an Eclipse component, the Eclipse component must have an action handler registered with the property broker. Whenever an action is invoked on the JSR168 portlet, the Portlet Container determines the action, and then fires a propertyChanged event through the property broker. In turn, the property broker determines the corresponding wire and invokes the action handler of the target Eclipse component.
RCP component view to portlet

When communicating information from an RCP component to a portlet, which is between an Eclipse component and a JSR168 portlet, every time an action is invoked on an Eclipse component, the component initiates a propertyChanged event through the property broker. The broker then determines the wire and invokes the processAction method of the target JSR168 portlet.

RCP component to RCP component

When an RCP component sends information to another RCP component through the property broker and wiring mechanism, and every time an action is invoked on a source RCP component, the component initiates a propertyChanged event through the property broker. The broker, in turn, determines the corresponding wire and invokes the action handler of the target RCP component.

These four communication schemata are the ones to be used. Nonetheless, there are other ways to intercommunicate two components in Lotus Expeditor, such as broadcasting and Click-to-Action (C2A). You will find more information about these additional communication features in Chapter 5, “Component communication” on page 85.

**Enabling two components to communicate**

If, for example, we already have designed and developed two magnificent components, and we want to make them cooperate by allowing one of the components, for example, the source component, to send some information to the other one, the target component, we must follow through some steps in order to enable those components to interact. An in-depth description of everything needed to accomplish this can be found in Chapter 5, “Component communication” on page 85, but the following gives you an overview in order to get a bird's-eye view of the whole process:

- Declare actions and properties for the source component in a WSDL file.
- Declare actions and properties for the target component in a WSDL file.
- Register the WSDL file of the source component with the property broker.
- Register the WSDL file of the target component with the property broker.
- If the target component is not a JSR168 portlet, register the action handler with the property broker using the PropertyBrokerAction extension.
- Modify the source component in order to publish properties.
- Code the action triggered by the target component. If the target component is a JSR168 portlet, you must modify the processAction method, or use an action handler.
Wire the source component to the target component. Note that wires can be defined and deployed to the Lotus Expeditor client platform in two ways. In a portal-managed environment, you can take advantage of the Portlet Wiring Tool. In a non-portal-managed environment, you must wire both components by using the PropertyBrokerWire extension point.

**Additional topics on inter-component communication**
Lotus Expeditor also leverages the access control capabilities by enabling the declaration of public and private wires.

Another feature imported from WebSphere Portal server is the capability to define cross-page wires, that is wires which can trigger an action in another page. However, the only way to enable this feature is to use the Portlet Wiring Tool in order to define the declared actions as global.

Finally, another special feature of the property broker mechanism is the possibility to cascade components in a communication flow and accomplish a chain effect by taking input properties and posting output properties.
Utilities for developing composite applications

In general, you can divide the development of composite applications into two phases. Phase one deals with individual component development that uses the development tools appropriate for that person and environment. Typically software engineers are responsible for this phase. The second phase is more related to business processes and a different set of people, who usually do not have software development skills. Because of this, the utilities for assembling composite applications have a different target group. They are aimed toward the business person.

This chapter provides an overview of available utilities for assembling composite applications. The following utilities are available:

- **Composite Application Editor**
  This is part of Lotus Notes 8 and provides support for portal and NSF based composite applications. See Chapter 23, “Utilities for assembling composite applications” on page 597, for details.

- **WebSphere Portal tools**
  This is part of WebSphere Portal with support for portal based composite applications. The Portal tools are described in this chapter.
3.1 WebSphere Portal tools

WebSphere Portal provides a wide array of development tooling to create robust composite portlet applications. For example, portlet applications can be quickly developed using Rational® Application Developer and the Portal Toolkit, WebSphere Portlet Factory, among other tools. For the scope of this book, we discuss developing composite portlet applications using Rational Application Developer and the WebSphere Portlet Factory.

3.1.1 Developing composite applications

Rational Application Developer using the Portal Tools provides the capabilities of visually developing and unit testing of individual portlets or entire portal applications. In the context of composite applications, the toolkit also provides built-in features for composite application development with support for developing Java Server Faces portlets, Struts portlets, JSR-168 portlets, and IBM Portlet API portlets. Portal Tools provides additional features, such as developing portlet template and editing and development of themes and skins, among other features. Although these additional features are not covered in this book, because they are out of the scope of composite application development.

The Portal Tools contain an advanced feature set for developing cooperative portlets (composite applications) which provides a visual development environment capable of generating code for broker based data exchange between portlets using declarative programming with taglib and generating WSDL for inter-portlet communication.

It is important to note that there are limitations that exist when developing composite applications with the Portal Tools as related to wiring. Shown below are the general limitations that exist with the Portal Tools that are installed with Rational Application Developer 7.0.

- Wiring is supported for portlets on the same page and for Cross-page wiring on WebSphere Portal 5.1.x and 6.0. For cross-page wiring on WebSphere Portal v5.1.x, PK18538 needs to be applied to the server.
- Only same-API wiring is supported (IBM API-to-JSR 168 API wires are not supported by WebSphere Portal 5.1.x and 6.0)
- Only global wires are supported. Personal wires, which enable you to configure user-specific wiring, are not supported.

Installation

With WebSphere Portal 6.x, the Portal Tools are installed as a part of Rational Application Developer. With prior versions of WebSphere Portal (that is, V5), the
Portal Tools were packaged as the Portal Toolkit, which has since been converted for distribution with Rational Application Developer installation. Refer to Figure 3-1 for an illustration of the Portal Tools installation option as part of the Rational Application Developer installation.

![Install Packages](image)

**Figure 3-1** Portal Tools installation via Rational Application Developer 7.x installation

### Developing composite (cooperative) portlet applications

The Portal Tools within the Rational Application Developer environment make it easier to develop the various parts of the runtime artifact to enable inter-portlet communication. The tooling provides support in the following areas.

- Declaring a portlet to be a source portlet or a target portlet
- Defining the data to be published as an output property or received as an input property in the source portlet
- Adding a trigger action to a source portlet for Basic (JSR-168) and Faces portlets
- Consuming the data in the target portlet
- Binding portal actions to the property broker operations

To enable a cooperative source or target of a portlet application, right-click the portlet application and select source or target as shown in FIGURE. A wizard will
appear guiding you through designating the portlet as a source or target, or both, of information. It is also important to note that a particular portlet may have multiple sources defined and multiple targets defined.

![Image](image.png)

Figure 3-2 Enabling a cooperative source or target

By having gone through the wizard for enabling a source or target, a WSDL file is generated on-the-fly. The WSDL file now acts as a neutral layer, or medium, of communications with other portlet or non-portlet applications. A few other steps must be taken within the portlet code based on the actions you want to have taken when a message is placed or taken from the wire, which you will see in greater detail in Chapter 7, “Developing portlet components” on page 153.

**Testing and deploying**

Rational Application Developer also enables you to test portlet applications directly within the development environment via the WebSphere Portal Test Environment. Once a portlet application has been developed, the development environment tooling also provides the a wizard that will help you publish the portlet application directly onto an existing and running WebSphere Portal server.

As shown in Figure 3-3, you can deploy directly to the portal or through another selection, can run on a test server within the Rational Application Developer portal test environment.
3.1.2 WebSphere Portlet Factory Designer

In the past, developing portlets, Web services and composite applications required expertise in both application development and portal technology. IBM WebSphere Portlet Factory Designer provides a portlet development environment for rapidly creating and maintaining customized portlets. This software development environment plugs into IBM Rational Application Developer or Eclipse 3.1 environment and allows you to build portlets by defining
a sequence of highly adaptive, reusable software components called builders. Developers begin by assembling builders into models, which WebSphere Portlet Factory software then interprets the model to dynamically generate the application code, Java Server Pages (JSPs), Java classes and XML documents, as well as the low-level artifacts that make up the portlet application. This allows developers to automate the process of building dynamic portlets, instead of explicitly coding each portlet. Additionally, WebSphere Portlet Factory Designer tool plugs seamlessly into IBM Rational Application Developer software and the open source Eclipse integrated development environment (IDE).

The User Interface
WebSphere Portlet Factory software consists of several objects that create the makeup of the tooling. These components work together or in progression to create a portlet application:

- **Builders** - Builders are software automation components that capture design intelligence and automate the creation of code. Builders have wizard-like user interfaces that provide a method to quickly develop portlets and services.

- **Model** - A model functions as a container that holds the ordered list of builders.

- **Profile** - A profile contains a set of parameters that vary the way an application behaves. It feeds values into a builder based on user identity or other contextual information, such as language. Using profiles, you can automatically generate different variations of a portlet (from the same model) for different users, customers or situations.

- **Regeneration** - Used to automatically regenerate the portlet code from the model. This is particularly used when a model has been modified after code has already been generated. It is important to note that the regeneration that occurs is based on a profile, that is, selecting sampleProfileA will generate code based on sampleProfileA's configuration; sampleProfileB will generate code based on sampleProfileB's configuration; and so on.

- **WebApp** - The Web application (WebApp) is a profile-specific instance of a portlet application that is dynamically created by the WebSphere Portlet Factory regeneration engine. Each builder, when called during regeneration, creates the artifacts that make up this WebApp, or run-time portlet application, such as pages, forms, variables, Java objects and methods. The regeneration engine creates the WebApp by regenerating a model with a unique instance of profile data.
Developing composite (cooperative) portlet applications

The Portlet Factory Designer tool allows you to create composite (cooperative) portlet applications using builders. With these tools, there are two ways to setup inter-portlet communications.

- Using cooperative Portlets - this is available for portlets built to the JSR-168 specification, also called the IBM Standard Portlet API
- Using click-to-action portlets - this is available for portlets built using the IBM Portlet API (deprecated).

Although both methods are available for use, it is preferred that cooperative portlets are used as the JSR-168 standard is utilized.

There are four general steps involved in building a simple composite portlet applications with the Portlet Factory Designer. These steps are listed below.

1. Create a cooperative portlet target model
2. Create a cooperative portlet source model
3. Send cooperative portlet events to the target portlets

Figure 3-4  User Interface of IBM WebSphere Portlet Factory Designer
4. Deploy a cooperative portlet WAR to the WebSphere Portal server

Create a Cooperative Portlet Target Model

When developing a cooperative portlet application, develop a target portlet first. By doing so, you can define the type of data or events you want to receive. This approach will simplify your design decisions when developing source portlets. To develop a target portlet, add the following to your model:

1. Add a Cooperative Portlet Target Builder to your model and use this Builder to define the event.
   You must specify the Event Name, Input Name, Input Type, Input Type Name, Namespace and Caption inputs. It is also important to remember that the Input information in the Cooperative Portlet Target Builder must match the Output information specified in the Cooperative Portlet Source Builder of any source portlets that want to interact.

2. Next add an Event Handler Builder to the model. The Event Handler will perform an action when an event occurs. In this case, you want to make sure that the Event Name input matches the Event Name input from your Cooperative Portlet Target Builder to ensure that the event will trigger the action.
   Note: Prior to specifying the Event Handler Builder, you should develop the method or page that the Event Handler will execute. This will allow you to select the event from the Action drop-down list in the Event Handler Builder Call Editor, rather than typing it in manually.

3. Finally, add a Portlet Adapter Builder to the model.
   This Builder call provides your model with a portlet title so that the portlet can be added to a page in WebSphere Portal. Since cooperative portlets were designed to function in a WebSphere Portal environment, it is important to remember this simple, but important step.
Create a Cooperative Portlet Source Model

Once you have developed a cooperative portlet target model you can develop a source portlet. To develop a source portlet, add the following to the portlet model:

1. Add a Cooperative Portlet Source Builder to the model and use this Builder to add either a C2A Single Action menu, or Property Broker link to a specified location on a page in your model. Note: The Property Broker Link functionality is available only for JSR-168 portlets. C2A menu functionality is available only for non JSR-168 portlets.

Make sure to specify the page location in your model on which the menu or link will appear, and specify the value to be passed to the target portlet. It is also important to remember that the Output information in the Cooperative Portlet Source Builder must match the Input information specified in the Cooperative Portlet Target Builder of your target model.
2. Add a Portlet Adapter Builder to the model.

This Builder call provides your model with a portlet title so that the portlet can be added to a page in WebSphere Portal. Since cooperative portlets were designed to function in a WebSphere Portal environment, it is important to remember this simple, but important step.

**Send Cooperative Portlet Events to Target Portlets**

Since cooperative portlets use the WebSphere Portal Property Broker technology, sending events to multiple target portlets is as easy as sending events to a single portlet. To send events to multiple portlets, you need not change your source portlet. To send an event to multiple portlets:

1. Add additional cooperative portlet targets, with the same Input Type information, to a portal page.

When broadcasting to multiple portlets, remember the outputs type information in the Cooperative Portlet Source Builder must match the input type information specified in the Cooperative Portlet Target Builder.

**Testing and Deploying**

Remember to rebuild (and redeploy) the portlet WAR after adding the Cooperative Portlet Builders to your model. You cannot add Cooperative Portlet Builders to your portlet models, save them, and then run portlets that exhibit the new functionality. This is because cooperative portlets describe target and source information using a WSDL definition file. The WSDL files must be added to the portlet WAR file prior to deployment. Rebuilding the portlet WAR will add the WSDL definitions and associated resource text files to it.

Note: If you have previously built and deployed the portlet WAR without cooperative portlets functionality, the portlet WAR must be removed and re-deployed for cooperative portlets features to function properly. You must also re-deploy the war if you make any changes that cause the generate WSDL file to change, or if you make a change to any related resource strings (that is, caption text).

Note: When using cooperative portlets with the Property Broker actions (JSR-168), you must use the WebSphere Portal Portlet Wiring Tool to connect the source and target portlets after deploying the WAR and placing your cooperative portlets on a portal page.

To re-deploy a portlet WAR to which cooperative portlets features have been added:

1. Develop or modify the model representing the portlet.

2. Modify the project and de-select the *Automatically Deploy Portlet WAR to WebSphere Portal* option.
3. Log in to the Portal and manually uninstall the Portlet WAR using the portal server's administration tool.

4. Modify the project and check the Automatically Deploy Portlet WAR to WebSphere Portal option. This will re-deploy the Portlet War automatically. Subsequent changes to the cooperative portlets models can then be deployed to the portal server using the Designers automatic deployment features.

If auto-deployment is NOT enabled in the project, portlet deployment steps are:

a. Develop or modify the model representing the portlet.

b. Uninstall the Portlet WAR using the portal server’s administration tool.

c. Use the Designer's Rebuild WAR > Portlet WAR right-click Project Menu choice to rebuild the WAR.

d. Manually install the Portlet WAR using the portal server's administration tool.

3.1.3 Workplace Dashboard Framework

Workplace Dashboard Framework is a flexible tool for rapidly building role-based dashboards on top of a service-oriented architecture. Workplace Dashboard Framework augments the capabilities of WebSphere Portlet Factory, adding dashboard-specific features, such as an alerting Module, hi-fidelity charting, flexible filtering technology, and a set of dashboard-specific design components called Builders.

Workplace Dashboard Framework provides tooling directly through the Portlet Factory Designer interface tooling. In fact, both tools coexist in the same development environment.
Installing IBM Portlet Factory Designer into existing development environment

Key components of Workplace Dashboard Framework

Workplace Dashboard Framework contains the following key components as part of its design.

- Dashboard-specific Builders. Workplace Dashboard Framework includes a set of Builders (reusable software automation components) designed to facilitate the creation of dashboard portlets. These Builders automate the most common dashboard design patterns and use cases. For example, Builders are included that support common dashboard features such as: status display using a dial chart, data summary and drilldown functionality, data hierarchy drilldown functionality, and data filtering functionality.

- Business User Configuration. Workplace Dashboard Framework enables you to extend the configuration of portlets to business users, including portlet look-and-feel, data, thresholds, process flow, and so on. With this capability, you can increase the flexibility of your dashboards.

- Hi-Fidelity Charting. Workplace Dashboard Framework enables the creation of dynamic, configurable charts. The product includes support for a wide range of chart types (for example, Pie, Dial/Gauge, Bar, Line) and file formats (for example, SQF, PNG, JPG, BMP).
Alerts Module. Workplace Dashboard Framework includes a robust Alerts Module which proactively alerts users, in real time, to important issues and events that affect their business. By having alerting as a core feature of the dashboard, users can quickly take action on problems or opportunities as they arise, rather than having to dig through numerous reports, data sources, or applications to find issues and opportunities.

There are three components to the Alerts Module. The first is the alerting engine which manages and fires the alerts to the appropriate people and through the correct channels – whether that is through the portal, by e-mail, or to a pager. The alert engine supports a number of different types of alerts. One type of alert is the basic threshold alert – which sends an alert if a certain metric falls above or below a threshold. For example, in a sales dashboard, you might want to trigger an alert to the sales executives if a hot opportunity has been stuck in the negotiation stage for more than 90 days.

The second component of the alerting functionality is the portal-based administration console that enables administrator to define alerts.

Flexible Filtering Capabilities. Workplace Dashboard Framework allows you to quickly create filters that enable users to dynamically filter the data in their dashboard portlets based upon criteria such as region, dates, product, brands, and so on. Filtering works hand-in-hand with the Dynamic Profiling technology built into WebSphere Portlet Factory. As a result, you can create filters that adapt according to the user’s characteristics and role.

### 3.1.4 Assembling composite applications using portlet wiring tool

The portlet wiring tool helps you create wires between multiple portlets within a page in addition to helping you create cross-page wires to portlets on other pages in the portal.

A wire allows two or more portlets on the page to transfer information so than an action or click in a source portlet automatically generates an action in the target portlet. As a result, wired portlets can be updated simultaneously when information is retrieved from the wire(s).

This tool can be used to perform the following steps:

1. Adding a wire
2. Deleting a wire
3. Setting-up global actions (cross-page wiring)

For example, to add, delete, or setup cross-page wiring using this tool, you can select portlet sources, targets, and pages the wires communicate with as shown in Figure 3-6.
Components of the portlet wiring tool

To add a wire using the Portlet Wiring tool the following information will be needed. An example of the tool is shown in Figure 3-6.

- **Source portlet** - The name of the portlet sending the information to other portlets on the page. You can only select portlets on the page that provide information for other portlets.

- **Sending** - This field identifies the type of information that the portlet is capable of sending.

- **Target page** - This is the page containing the receiving action. The drop-down list contains only peer pages that have defined to receive global actions. If the desired page is not listed, select More and select the desired page or search for the target page.

Note: If a target page with no global receiving actions is selected for cross-page wires, you will not be able to select target portlets or receiving actions.

- **Target portlet** - The portlet receiving the information from the source. Only portlets that are capable of processing the Sending information are shown.

- **Receiving** - The action that the portlet can perform after receiving the information. Only actions that are capable of handling the selected information are shown.

**Note:** If you have administrative privileges, you can set the wires for all users of the page by selecting Public as the wire type. By default, all wires are created for personal use.
Once you have made changes for either adding or deleting a wire in the Portlet Wiring Tool, click the Add new wire icon. The new wire appears in a list of all wires on the page each time you edit the wires for the page. At this point the wires are enabled.

**Portlet wiring tool and relationship to WSDL files**

It is also important to note that the wires are made visible to the Portlet wiring tool based on the WSDL file definitions created in source / target portlets. For example, if a source wire action has been defined in the source portlet WSDL, file, but a target wire action is not defined in the target portlet’s WSDL file, then the portlet wiring tool will show a target wire action available for selection. For example, shown in Example 3-1 and Example 3-2 on page 67, you will see sample source and target WSDL files that will be used later in the scenarios.

First, let us review the source portlet sample WSDL code as shown in Example 3-1.

**CustomerIdentification.wsdl**

To relate the following WSDL sample to the Portlet Wiring Tool, the correlations are described below. In this file, the portlet wiring tool recognizes that the name of the source portlet parameter is **outputCustID** as shown in Example 3-1. In the Portlet Wiring Tool, this property appears in the “Sending” field as shown in Figure 3-6. The source portlet name does not appear in the WSDL as the WSDL is directly related to the source portlet name via the portlet.xml portlet descriptor.

Also shown boldface in Example 3-1 is “CustID”. This message type name is what is used to link the various sources and targets. In the target WSDL, file, the text must match or the Portlet Wiring Tool will not be able to link the source to the correct target(s).

---

**Example 3-1  CustomerIdentification.wsdl located project’s /wsdl folder**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:portlet="http://www.ibm.com/wps/c2a"
  <types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://userreservationsproject">
      <xsd:complexType name="CustID">
        <xsd:restriction base="xsd:string"/>
      </xsd:complexType>
    </xsd:schema>
  </types>
</definitions>
```
The wire definition is shown in Figure 3-7.

Figure 3-7  Property of source portlet
Next, review the target portlet sample WSDL code as shown in Example 3-2.

**CustomerReservations.wsdl sample file**

The Portlet Wiring Tool recognizes the available targets based on the matching the name of the WSDL message type, in this case, “CustID” for the target portlet of CustomerReservations. As noted earlier, the CustomerReservations target is listed as a target as an input property is defined for the message type “CustID”. For the target portlet, CustomerReservations, the drop-down menu in the Portlet Wiring Tool provides an option for selection named “CustIDAction, inputCustID Property”. CustIDAction is the action name of the target portlet WSDL and the inputCustID is the input parameter being received by the target portlet as shown in.

**Example 3-2  CustomerReservations.wsdl located project's /wsdl folder**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:portlet="http://www.ibm.com/wps/c2a"
  <types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://userreservationsproject">
      <xsd:simpleType name="CustID">
        <xsd:restriction base="xsd:string"/>
      </xsd:simpleType>
      <xsd:simpleType name="ResID">
        <xsd:restriction base="xsd:string"/>
      </xsd:simpleType>
    </xsd:schema>
  </types>
  <message name="CustID_Request">
    <part name="CustID_Input" type="tns0:CustID"/>
  </message>
  <message name="ResIDAction_Response">
    <part name="ResID_Output" type="tns0:ResID"/>
  </message>
  <portType name="CustomerReservations_Service">
    <operation name="CustomerReservations">
      <input message="tns:CustID_Request"/>
    </operation>
    <operation name="CustomerReservations0">
      <output message="tns:ResIDAction_Response"/>
    </operation>
  </portType>
</definitions>
```
The wire definition is shown in Figure 3-8.

Figure 3-8  Property of target portlet
Components in composite applications

In this chapter, we introduce you to the types of components that can be used in Lotus Expeditor composite applications. We explain the component types that are available today and provide a future outlook to things which are under development and will be ready for component developers at a later time.

This chapter describes these components:

- Local Eclipse components. This provides a short overview on the concepts of an Eclipse plug-in.
- Portlets. This introduces the issues to consider when bringing your portlet applications to the Lotus Expeditor’s client container.
- Web applications. This provides details of the client’s Web container.
- Future types of components. This explains what potentially could be used as components in the future.
4.1 Overview

Unlike many other technologies, such as Enterprise JavaBean (EJB™) and many others, the term *component* does not refer to something that is declared as a standard. The term component is used to define any arbitrary piece of software that can be somehow packaged and reused in one way or another. Components do not necessarily provide a user interface, however, some of them do.

To narrow down the specifications of a component in the context of Lotus Expeditor, a few more assumptions must be made. Thus, components get a more granular specification.

**Declaring reference information**

First of all, to allow components to be managed by a platform, they must follow certain rules. For example, they need to know how they will describe themselves to the managed platform. Since Lotus Expeditor is based on Eclipse technology and therefore contains an Open Service Gateway initiative (OSGi) layer, the concept of OSGi bundles is probably the best way to illustrate this.

OSGi defines how a piece of software or a collection of pieces of software can be packaged. It also defines how the various pieces within this package make themselves known to the platform. Eclipse-savvy readers will recognize the concept of *bundles*, sometimes also referred to as *plug-ins*. Bundles can be bundled again to form a somewhat higher level of collection known as a *feature*.

**Note:** In Eclipse a bundle is a plug-in.

**How software components execute**

In most cases, software components running on Eclipse based platforms are Java programs written to the Plugin-Development-API of Eclipse, thus, using Standard Widget Toolkit (SWT) or SWT/JFace as their user interface. If this is the only legitimate component, it excludes all the Web and portal application developers who wrote fine code to be used on another platform.

To broaden the scope of applications types that can be executed, Lotus Expeditor comes with a complete Web container implementation that allows you to run servlets and JavaServer™ Pages™ (JSPs), almost to the same extent as on a regular Web Application Server. Furthermore, the concept of portal applications have been also brought to Lotus Expeditor which provides a way to take a portlet application that was intended to run on a standard JSR168 compliant portal server (such as WebSphere Portal), and run it on the Lotus Expeditor client platform.
How components communicate
To form a real composite application, its components within must be able to communicate with each other. This is not an uncommon desire; there are probably hundreds of ways of how pieces of code send data back and forth.

The important thing to remember in a composite application is that independent software vendors are supposed to write independent components that, later on, can be assembled into a composite application to form a business service. Therefore, none of the component developers can potentially anticipate this when developing the component. To achieve such a degree of independency, a standardized way of communication is used for component to component communication in Lotus Expeditor. Its origins are based on Web Services and its implementation comes from the WebSphere Portal product. Chapter 5, “Component communication” on page 85 elaborates on this particular subject in greater detail.

4.2 Local Eclipse components

Is not anything that runs inside Eclipse something local? That is a question of the perspective you are taking. Eclipse bundles can use all sorts of ways to run and display themselves. When it comes to integrating existing applications into Eclipse, people have found ways to bring their software pieces into Eclipse.

For applications intended to run locally on the operating system, a mechanism of re-parenting the window of the application was developed. This procedure allows an Eclipse-based plug-in to launch the local application, and forces its output into an Eclipse-based view. For non-Eclipse-based Java applications, Eclipse (since version 3.0) provides an Abstract Window Toolkit (AWT)/Swing-bridge. This gives developers a chance to integrate their AWT/Swing based applications without hardly touching the code.

In this document, when we mention local Eclipse components, we refer to them as Rich Client Platform (RCP) components, and we assume that the component has the following characteristics:

- It is written in Java (at least to a large extent)
- It uses SWT based widgets and dialogs
- It is compliant to OSGi’s specification of a bundle
4.3 Client service portlets project

One of the new features now available in Lotus Expeditor is the support for portlet applications. Portlet applications provide a consistent view of enterprise applications. The combination of portlets generate a portal view. A portal view can be accessed, for example, from:

- A portal server such as WebSphere Portal by using a Web browser
- An RCP such as Lotus Expeditor

The portal allows the administrator to define specific sets of applications, which are presented to the user in a single page context (browser) or a single perspective (RCP).

**Note:** The portlets themselves are simple views of existing or new enterprise application content. A portlet is a complete application having multiple states and view modes, plus event and messaging capabilities.

Portlets run inside the portlet container, and, in a similar way, a servlet runs on an application server. The portlet container provides a runtime environment in which portlets are instantiated, used, and finally destroyed. Portlets rely on the portal infrastructure to access user profile information, participate in window and action events, communicate with other portlets, access remote content, look up credentials, and store persistent data.

The user interacts with portlets through *action URIs*. By clicking on a link inside of the portlet the user performs certain actions, for example, changing the display view. Internally the portlet stores persistent information about its current state. A portal view aggregates many different portlets at the same time, which is an important issue to consider when you design and work with portlet applications.

4.3.1 Lotus Expeditor portlet support

Standard portlets (JSR168) are a specification from the Java Community Process for the standardization of portlets. The specification was developed to provide interoperability for running portlets on any vendor's implementation of the JSR168 portlet container.

WebSphere Portal v6.0 provides a runtime environment for both the IBM Portlet application programming interface (API), which is based on org.apache.jetspeed.portlet interfaces and JSR 168 or compliant portlets. However, the Lotus Expeditor platform supports only portlet applications, which are developed to conform to the JSR 168 specification.
The Expeditor Toolkit provides portlet development tooling components that allow users to develop and debug client services portlet applications based on the OSGi framework, as well as converting existing Java 2 Platform, Enterprise Edition (J2EE) server based portlet applications to run on the Expeditor platform.

The portlet development tools also allow users to wire and aggregate portlet applications. A portlet application bundle can be developed using many of the same portlet development tools provided by the IBM Rational Software Development platform.

The portlet client service API and the portal portlet API basically follow the same rules. From the portlet programmer view, there are no differences related to the portlet code. However, the portlet package must include a plugin.xml file in order to be executed in a Lotus Expeditor environment.

As illustrated in Example 4-1, the plugin.xml file must always include the extension point com.ibm.eswe.workbench.WctWebApplication.

**Example 4-1  Sample WctWebApplication extension point definition in plugin.xml**

```xml
<extension id="DEMO Wire"
    point="com.ibm.eswe.workbench.WctWebApplication">
    <WEBApplication DisplayName="DEMO Wire Send Message">
        <WebApplicationUrl local="true" secured="false"
            url="/DemoWire/DemoWire/default/ver=1.0/mode=view"/>
        <BrowserConfiguration showAddressbar="true" showBookmark="false"
            showHistory="true" showHome="false" showPrint="false"
            showToolBar="true"/>
    </WEBApplication>
</extension>
```

The following information is included in the extension point definition:

- **WEBApplication DisplayName.** This is the name of the application in the Expeditor launch menu when the user wants to invoke a portlet application.

- **WebApplicationUrl.** This is the portlet URL. It is information added to the context root (com.ibm.pvc.webcontainer.application extension point plug-in) and the portlet name. You can also define the initial portlet mode.

- **BrowserConfiguration.** In this tag you can define the embedded Web browser options when showing the portlet application. For example, you can enable the browser address bar which is very useful when developing portlets. You can also enable the print button, home page and bookmark in the embedded browser.
Another important extension point is `com.ibm.pvc.webcontainer.application`. This extension point defines the portlets in this plug-in. As shown in Example 4-2, the supported portlet modes are also defined here.

**Example 4-2  Sample webcontainer.application extension point and portlet definitions**

```xml
<extension point="com.ibm.pvc.webcontainer.application">
  <contextRoot>/.DemoWire</contextRoot>
  <contentLocation>WebContent</contentLocation>
  <actionClass>javax.portlet.PortletSessionUtil</actionClass>
  <portletOptions>
    <portlet>
      <portlet-name>DemoWire</portlet-name>
      <supports>
        <portlet-mode>view</portlet-mode>
      </supports>
    </portlet>
  </portletOptions>
</extension>
```

### 4.4 Client service Web project

The Lotus Expeditor platform supports Servlet v2.4 and JSP™ v2.0 Web applications and Servlet v2.3 and vJSP 1.2 Web applications. Web applications targeting the Lotus Expeditor platform are called Client Services Web applications.

Since components in the Lotus Expeditor platform are called bundles, a Web application targeting this platform is also referred to as a Web Application Bundle (WAB).

A primary difference between a WAB and a Web component, which is deployed to run on a WebSphere Application Server run time, is that the WAB file must be a valid OSGi bundle. The Expeditor Toolkit automatically handles many of the bundle specific details.

It is also possible to develop the Web application through a dynamic Web project, and subsequently test run it on the Lotus Expeditor platform. Using the provided WAB utility, it is also possible to transform an existing Web application archive (WAR) file into a WAB suitable for running on the Lotus Expeditor platform.
As illustrated in Example 4-3, the plugin.xml you must include the extension point com.ibm.eswe.workbench.WctWebApplication.

Example 4-3  Sample WctWebApplication extension point

```
<extension id="hello" point="com.ibm.eswe.workbench.WctWebApplication">
  <WEBApplication DisplayName="ITSO List Contracts">
    <WebApplicationUrl local="true" secured="false">
      url="/com.ibm.itso.compapp.carrental.listcontracts"/
    </WebApplicationUrl>
    <BrowserConfiguration showAddressbar="false" showBookmark="false" showHistory="false" showHome="false" showPrint="false" showToolbar="false"/>
  </WEBApplication>
</extension>
```

The following information is included in the extension point definition:

- **WEBApplication DisplayName**: this information is the name in the launch menu to select this application.
- **URL**: This defines the Web context root. You can also specify the name of JSP and servlet component. When you do not specify the name of the resource, Lotus Expeditor gets the default page name from the `<welcome-file-list>` tag in the web.xml file.
- **Browser Configuration**: in this tag you define the Web embedded browser to show the application. You can also enable the browser address bar (this option is useful when you are testing the application). In addition, you can optionally enable the print button, home page and bookmark in the embedded browser.

Additionally, the extension point where the Web application (servlet) is defined must be included in the plugin.xml file. Example 4-4 shows a sample extension point for the com.ibm.pvc.webcontainer.application.

Example 4-4  Sample webcontainer.application extension point and servlet definitions

```
<extension point="com.ibm.pvc.webcontainer.application">
  <contextRoot>/com.ibm.itso.compapp.carrental.listcontracts</contextRoot>
  <contentLocation>WebContent</contentLocation>
</extension>
```
4.5 Remote portlets

The Web Services for Remote Portlets (WSRP) standard allows the integration of remote portlet content into a user portal, simplifying the effort of portal administrators in selecting and including a rich variety of remote content and applications from external sources into their portals without any programming effort.

Some terms used in WSRP are:

- **Producer**: is a portal that provides one or more portlets as WSRP services and makes them available to consumers.

- **Consumer**: is a portal container that invokes WSRP services from producer portals and integrates them to be used locally, for example, Lotus Expeditor.

- **Remote portlets**: portlets that have been published as WSRP services by a producer portal.

As illustrated in Figure 4-1, WSRP components in composite applications access remote portlets from a producer portal.

*Figure 4-1  Accessing remote portlets as components*
4.5.1 WSRP services

WSRP services are presentation-oriented, providing the application data and the presentation layer of the portlet. On the other hand, Web Services are only data-oriented, providing the application data as shown in Figure 4-2.

![Figure 4-2 A comparison of Web Services and WSRP](image)

WSRP standard defines a set of interfaces provided by producer portals to be exposed to the consumers. These WSRP interfaces are described in a Web Services Description Language (WSDL) document accessed by the consumer. In Lotus Expeditor composite applications, the consumer is the Portlet Viewer WSRP component. The WSDL file provides information to the consumer component with information about how to bind to the producer portal.

The interfaces are:

- Service Description: is a self-description of the producer, its capabilities and its available portlets. It provides further information to consumers about the producer and its properties. This interface is mandatory.
- Markup: is an interface to get and process interactions with markup fragments. It allows you to obtain portlet markups from the producer and submit portlet requests from the consumer. This interface is mandatory.

- Portlet Management: defines operations for cloning, customizing and deleting portlets. It allows you to customize and manage remote portlets preferences in consumer portals. This interface is optional.

- Registration: an optional interface for consumer registration in a producer portal. It allows the producer to identify each consumer.

**Note:** This optional interface is not implemented in WebSphere Portal.

Example 4-5 illustrates a sample WSDL file generated by the WebSphere Portal server tools (producer). In this sample file, `<portal>` represents the domain name of the portal server running the actual portlet. Also, the location of this WSDL file (URL) must be specified in the plug-in extension point as shown in Example 4-6 on page 79.

**Example 4-5   Sample WSDL file for WSRP**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:oasis:names:tc:wsrp:v1:wsdl"
xmlns:bind="urn:oasis:names:tc:wsrp:v1:bind"
xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
  <import namespace="urn:oasis:names:tc:wsrp:v1:bind"
      location="/wsrp_v1_bindings.wsdl"/>
  <wsdl:service name="WSRPService">
    <wsdl:port binding="bind:WSRP_v1_Markup_Binding_SOAP"
        name="WSRPBaseService">
      <soap:address location="/wsrp/WSRPBaseService"/>
    </wsdl:port>
    <wsdl:port binding="bind:WSRP_v1_ServiceDescription_Binding_SOAP"
        name="WSRPServiceDescriptionService">
      <soap:address location="/wsrp/WSRPServiceDescriptionService"/>
    </wsdl:port>
    <wsdl:port binding="bind:WSRP_v1_PortletManagement_Binding_SOAP"
        name="WSRPPortletManagementService">
      <soap:address location="/wsrp/WSRPPortletManagementService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Note: This optional interface is not implemented in WebSphere Portal.
4.5.2 Lotus Expeditor WSRP support

Lotus Expeditor provides a portlet viewer for remote portlets. The portlet can be used to view a remote portlet through the WSRP feature in Lotus Expeditor. In the portal-managed environment, you can deploy a remote portlet as part of a composite application. For details see 14.6.3, “Placing a WSRP portlet viewer” on page 361. As illustrated in Example 4-6, the WSRP portlet viewer instances are defined and contributed using the Portlet Viewer extension point com.ibm.rcp.portletviewer.WsrpPortlets.

Example 4-6 Sample extension point definition for WSRP portlet viewer

```xml
<extension point="com.ibm.rcp.portletviewer.WsrpPortlets">
  <wsrpData entityId="com.ibm.itso.compapp.weather.wsrp"
    wsrp_wsdl="http://<portal>/wps/wsdl/itso-weather2.wsdl"
    handle="5_8GUNUKG100CKC02EESU1TF10V0" need_client_clone="true"
    isSecured="false">
  </wsrpData>
</extension>
```

4.6 Future potential components

The components listed so far all have one thing in common, you have to develop them in Java and their code is locally run by the virtual machine that Lotus Expeditor runs in. There are other constructs for developing applications, which use other ways to execute their code, thus allowing a higher level of abstraction for the developer. These tools are commonly called Builders or Designers. They usually come with a special run time that needs to be deployed in order to run code that has been developed with it.

Three products of this type exist in the IBM software group today that make use of such a concept and are intended to be enhanced in a way that they can be used to create components for Lotus Expeditor in one of their future versions.

**IBM Lotus Notes Domino**

Hannover is the code name for the next release of IBM Lotus Notes. In Hannover, customers will see step-function improvements in the Lotus Notes
client's core capabilities, such as e-mail, calendaring and scheduling, and contact management. There will also be new and innovative capabilities including activity-centric collaboration, which can dramatically increase user productivity, and support for composite applications, which allows developers to combine their existing Lotus Notes applications with line-of-business application components to solve specific business problems.

**Hannover as a Lotus Expeditor application**

The first thing Lotus Notes users will notice in Hannover is a fresh, new user interface, but the changes go beyond cosmetics. Hannover, as the current IBM Sametime 7.5 client, is actually built based on Lotus Expeditor. Thus, it inherits all the features that Lotus Expeditor offers as a client platform.

**A new class of open, role-based applications**

Another defining capability of the Hannover release is the support for composite applications, that is, applications, such as those being delivered via portals today, which combine components from multiple applications into a single, role-based work environment. For example, a composite application for sales teams might combine a standard Lotus Notes collaborative application with components from sales force automation (SFA), customer relationship management (CRM) and
order-entry applications. Figure 4-3 shows a potential Hannover composite application.

![Figure 4-3 Potential Hannover composite application](image)

Hannover allows users to connect multiple applications into a single, integrated composite application.

**IBM Lotus Component Designer 6**

IBM Lotus Component Designer 6 is an innovative, easy-to-use standards-based development tool that script developers, Visual Basic® developers, IBM Lotus Domino® application designers, and others can use to easily create components for IBM WebSphere Portal 6 applications. It supports a service-oriented architecture (SOA), and is ideal for composite application development and extending existing applications with collaborative components.

IBM Lotus Component Designer 6 is easy to learn. You can build secure, flexible, XML document based applications, such as expense reports, project management, document approvals, blogs, or change requests in less time than with other technologies and tools. Reusable user interface (UI) controls can be created and added to the Component Designer palette. You can use the same
easy metaphor to access data in Java Database Connectivity (JDBC™) or Domino data sources, or Web Services.

- The Component Designer 6 tool (Integrated Development Environment (IDE)) is installed on Eclipse v3.2. You use the tool to create and preview component pages and deploy components.
- The Component Designer 6 run time is installed on WebSphere Portal v6.0. You must have Portal v6 to run the deployed Component Designer components.

Key features in IBM Lotus Component Designer 6 are:
- Deploy applications to WebSphere Portal 6
- New IDE runs on Eclipse 3.2
- Work visually, or with a source code
- Create workgroups to manage projects
- Outline the view to access design elements
- UI to easily apply styles
- AJAX support
- Lightweight preview functionality
- Access external data and Web services: Domino, relational databases
- Relational query builder
- User-defined custom components
- Support for server side script libraries
- Adds inter-component communication for composite applications
- Ability to create multiple portlets in a single component
- Create multilingual components
- WebSphere Portlet Factory

**Note:** As of the time of writing this book, IBM Lotus Component Designer 6 is in its beta phase and Version 6 is not intended to support creating components that can be deployed to a Lotus Expeditor client. However, it is planned to have a future version of IBM Lotus Component Designer to support the creation of such components.

**WebSphere Portlet Factory**

IBM WebSphere Portlet Factory supercharges WebSphere Portal with tools and technology to rapidly create, customize, maintain, and deploy portlets. It enables developers of any skill level to construct, change, deploy, and maintain custom portlets, creating an SOA entry point for companies by delivering SOA-based portal applications.

IBM WebSphere Portlet Factory provides pre-built integrations for existing applications such as Lotus Notes and Domino, SAP®, PeopleSoft®, Siebel®, Databases, and Web Services.
**Builders**

At the core of the IBM WebSphere Portlet Factory Designer are software automation components called Builders. These Builders capture design intelligence and automate the creation of code. Similar to customizable robots in an assembly line, Builders perform specific automation tasks based upon inputs or parameters specified by developers. WebSphere Portlet Factory ships with over 165 Builders that automate a wide range of tasks, such as creating HTML from a schema or integrating with common back-end systems such as IBM Lotus Domino, SAP, Siebel, PeopleSoft, Web Services, and JDBC-compliant databases.

Builders have easy-to-use, wizard-like user interfaces, which speed up development, thereby masking the complexities of the underlying J2EE or portal APIs, and produce portlets that are SOA compliant. As a result, Builders increase developer productivity, reduce coding errors, and improve code quality.

Behind the scenes, a Builder is made up of a Java class that performs the appropriate automation task (such as creating the JSP for the button) and an XML document that defines the Builder’s characteristics. Since Builders are based on an open, extensible architecture, developers can easily create their own Builders to automate custom design patterns or to enforce compliance to company architectural and coding standards.

Builders take care of the repetitive programming tasks that developers typically do and redo by hand, such as wiring up data to presentations. The net result is that Builders free developers from mundane programming tasks, allowing them to focus on tackling the problems that are of more business value to the organization.

**WebSphere Portlet Factory Designer**

IBM WebSphere Portlet Factory includes an easy-to-use graphical tool for creating, viewing, and running portlets: the IBM WebSphere Portlet Factory Designer. The WebSphere Portlet Factory Designer provides simplified rapid application development of custom portlets for IBM WebSphere Portal that leverage existing enterprise applications, data and systems, including IBM Lotus Domino, SAP, PeopleSoft, DB2® and Web Services, among others. It does this without requiring J2EE development expertise, and without developers needing to implement and learn application and portal APIs by automating portlet development with reusable wizard-like components. It, like IBM Lotus Component Designer and Lotus Expeditor, runs on the Eclipse platform.
Component communication

This chapter describes how components can be prepared to participate in communication. We discuss how the necessary information to allow communication to happen is declared and how components have to be made communication-aware. In particular, in this chapter you find:

- How to create WSDL files defining the bits and pieces of communication
- How to update your code to allow for communication to happen
- How to create wires between your components to control communication
5.1 Overview

One of the most important characteristics in composite applications is the capability to take two components and wire them together for communication. Generally, a function like this only works satisfactorily if one of the following cases is available:

- The communication between the two components has been developed into the component. Thus, the components know of each other in advance during their development cycle.

- A neutral way of communication is used to separate the declaration of data items and the transport layer from the component's code.

The latter is the case in the Lotus Expeditor platform. Lotus Expeditor comes with a feature called the property broker, a technology that Expeditor inherited from WebSphere Portal.

Figure 5-1 shows an excerpt of the ITSO Car Rental sample application used in this book to demonstrate the capabilities of composite applications. There is a booking component on the left, it allows you to look up customer information. There is also a car selection component on the right side, it provides a way to search for available cars and select a specific model for the rental contract.

These two components ideally communicate with each other. In this sample scenario, if the customer information is updated, the preferred car size for this customer must be communicated to the car selection component in order to update its car list. If, in turn, a car is selected, the car details are transferred back to the booking component.
Let us further consider that the booking component is used worldwide in ITSO Car Rental, but there are different car selection components in different geographies where ITSO Car Rental does business. Then, instead of having to provide the internal details related to the booking component’s code, all it needs to be available for interfacing with the component, is a WSDL file. In this file, the input and output data items, which the components understand, are declared. All the rest, such as linking the components together and transferring the data back and forth, are taken care of by the property broker.

5.2 Declaring data items and actions

In order to be neutral between components, the declaration has to be specific enough to guarantee the proper transfer of data. On the other hand, it also needs to be flexible enough to allow the linking of components using a generic tool. Both issues have been achieved using a Web Services Definition Language (WSDL) file where this is described. Figure 5-2 gives a graphical representation of a sample WSDL file.

Figure 5-2  Graphical representation of a WSDL file
Starting at the left most column, a namespace has to be declared that is common to all type definitions. This example uses:


This is a namespace that IBM has reserved for its Click-to-Action (C2A) technology in WebSphere Portal. In this namespace, the data types are declared.

**Note:** The property broker in Lotus Expeditor supports data types that are of type java.lang.String. If you need to transfer more complex data, you have to make use of serialization or comparable mechanisms.

The middle column in Figure 5-2 contains two definitions:

- A message for each direction is defined. The data items are also defined by referencing the data type.
- These message definitions can then be used to define a port type, which is basically an empty operation defining the input and output messages.

The right column in Figure 5-2 brings it all together. By using the port type, it declares the action that can be called on, to inform the component of changes to the properties it accepts as input. It also declares the input and output variables with a name and caption that is readable for users.

Example 5-1 illustrates an example of a WSDL file that is used to declare the credit card authorization process.

*Example 5-1  WSDL file for credit card authorization process*

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="LoadURLInBrowser_Service"
    targetNamespace="http://www.ibm.com/wps/c2a"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:portlet="http://www.ibm.com/wps/c2a"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.ibm.com/wps/c2a"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <types>
        <xsd:schema targetNamespace="http://www.ibm.com/wps/c2a">
            <xsd:simpleType name="java.lang.String">
                <xsd:restriction base="xsd:string"/>
            </xsd:simpleType>
        </xsd:schema>
    </types>
</definitions>
```
<xsd:schema>
  
  <message name="receiveAuthMessage">
    <part name="Card Number" type="portlet:java.lang.String"/>
    <part name="Authorization Amount" type="portlet:java.lang.String"/>
  </message>

  <message name="responseAuthMessage">
    <part name="Authorization Result" type="portlet:java.lang.String"/>
  </message>

  <portType name="receiveAuth">
    <operation name="receiveAuthOperation">
      <input message="tns:receiveAuthMessage"/>
      <output message="tns:responseAuthMessage"/>
    </operation>
  </portType>

  <binding name="receiveAuthBinding" type="portlet:receiveAuth">
    <portlet:binding/>
    <operation name="receiveAuthOperation">
      <portlet:action name="receiveAuthAction" type="standard"
        caption="Credit Card Authorization"
        description="This will check for credit card authorization"
        actionNameParameter="ACTION_NAME"/>
      <input>
        <portlet:param name="CC Credit Card Number" partname="Card Number"
          caption="Customer's credit card number"/>  
        <portlet:param name="CC Authorization Amount" partname="Authorization Amount"
          caption="The amount that has to be authorized"/>
      </input>
      <output>
        <portlet:param name="CC Authorization Result" partname="Authorization Result"
          caption="Result of the authorization (true/false)"/>
      </output>
    </operation>
  </binding>
</xsd:schema>
Here is a more human readable interpretation of the contents of this WSDL file:

- Define a data type called java.lang.String in the namespace of:
  
  http://www.ibm.com/wps/c2a

  This has the restrictions of xsd.string.

- Define a message called receiveAuthMesssage that has two data items, Card Number and Authorization Amount, both of type java.lang.String.

- Define a message called responseAuthMessage that has one data item, Authorization Result of type java.lang.String.

- Define a port type name: receiveAuth, which has an operation named receiveAuthOperation and contains an input message of type receiveAuthMessage and an output message of type responseAuthMessage.

- Declare a binding called receiveAuthBinding of port type receiveAuth containing an operation of type receiveAuthOperation. Here, an action is declared with the name receiveAuthAction. For basic portlets, its type is standard. Its action name parameter is ACTION_NAME and it also has a caption and a description for human beings.

This action now declares the input and output data items it can accept. The input data items are called CC Card Number (of part name Card Number) and CC Authorization Amount (of part name Authorization Amount). The output data item is called CC Authorization Result (of part name Authorization Result).

When creating WSDL files, your development tool can be of great help to reduce the complexity of these files. Both Eclipse, when equipped with the Web Tools Platform (WTP), and IBM Rational Application Developer (RAD) provide a WSDL editor. This editor allows for easier editing of WSDL files. Figure 5-3 illustrates an example of what the WSDL editor looks like.

![Figure 5-3 WSDL editor in Eclipse and RAD](image-url)
5.3 Preparing components for intercommunication

Letting components participate in communication requires a few updates to the component's code. In most cases, these updates are very trivial, depending on how your component is designed.

In a nutshell, you must provide functionality to send data and you must provide an entry point for the property broker to call your component to let it know that new data has arrived that could be of some interest.

The next sections in this chapter outline what changes have to be implemented for sending and receiving data in portlets and Eclipse Rich Client Platform (RCP) components.

5.3.1 Portlet components

In this scenario, a very simple sample composite application is created to send a message and enable portlet communication between two portlet components. In Lotus Expeditor composite applications, this is accomplished through the implementation of the wired model.

Portlet components that are required to send a message and listen for message events must register their properties in the property broker. Figure 5-4 illustrates the sample scenario where two portlets communicate via the property broker.
The execution flow for this sample scenario has the following sequence:

1. Lotus Expeditor invokes the sample composite application with two portlet components (send and receive message portlet components). The user enters a message text in the send message component and submits the request as shown in Figure 5-5.
2. The `processAction()` method in the send message portlet component (source portlet) detects if this action is to publish a value to the property broker. This output property must be defined in the source portlet WSDL file.

3. The property broker obtains the parameter and identifies the target components. This information must be available in the `<wire>` tag of the `plugin.xml` file.

4. The `processAction()` method in the receive message portlet (target portlet) is invoked and receives the input property. Again, the input property must be defined in the WSDL file of the target portlet component.

5. In this sample scenario, the JavaServer Pages (JSP) in the target portlet reads the value and writes the message as shown in Figure 5-6.

![Figure 5-6 Receiving the message from the source portlet component](image)

This scenario has one string message. It is important to understand that you can create other kinds of messages and interact with other components.

**Note:** Other important detail is that in this sample scenario, if a message in entered in the target portlet, the same `processAction()` method will also handle the submit from the keyboard. That is, the action is processed in the same method whether it arrives from the keyboard or from the property broker.

**Sending a message from a source portlet component**

The implementation of a wire follows the same infrastructure for wiring standard JSR168 portlet applications. Properties are connected to each other through the creation of wires. Wires are used to connect an output property in a source component to an input property in a target component.

You also need to be aware that the data type of the two properties must be common to both components. In the sample scenario, the `wire_text` variable,
representing the business data, is used. As an option, this variable can be defined as a Java constant object as illustrated in Example 5-2.

**Example 5-2  Java constants**

```java
public static final String FORM_SUBMIT = "SendMessagePortletFormSubmit";
public static final String FORM_TEXT = "wire_text";
public static final String ACTION_NAME_PARAM = "ACTION_NAME";
public static final String SEND_MESSAGE_ACTION = "MessageWireAction";
```

Whenever the first component publishes the current value of the wire_text property, the property broker transforms the string to a property called wire_wsdl_text and sends a notification to the receive message portlet (target portlet) component. The target component receives the message, updates the page, and displays the results.

**The processAction() method in the source portlet component**

Generally, this method is invoked by the portal container during the event processing phase of the portlet to process an action event, such as a submit request from the keyboard.

**Note:** You can only publish an output property during the event processing phase of the source portlet life cycle.

In this scenario, as illustrated in Example 5-3, the first step in this method is to determine the kind of action received by obtaining its value from the request object using the getParameter() method. The text message is stored in the request object as an attribute and the WSDL file must indicate this to the property broker.

**Example 5-3  Process Action Method**

```java
public void processAction(ActionRequest request, ActionResponse response) throws PortletException, java.io.IOException {
    // send message to property broker
    String wiredText = null;
    String actionName = request.getParameter(ACTION_NAME_PARAM);

    if(SEND_MESSAGE_ACTION.equals(actionName)){
        wiredText = request.getParameter(FORM_TEXT);
        request.setAttribute(FORM_TEXT, wiredText);
    }
}
```
Since output properties can only be published during event processing, theportlet method doView() has no property broker responsibility and in this case its only function is to specify to the portal container what JSP should be called. This is illustrated in Example 5-4.

**Example 5-4  View mode processing in the source portlet component**

```java
public void doView(RenderRequest request, RenderResponse response)
throws PortletException, IOException {

    // Set the MIME type for the render response
    response.setContentType(request.getResponseContentType());

    // Invoke the JSP to render
    PortletRequestDispatcher rd =
      getPortletContext().getRequestDispatcher(JSP_PATH);
    rd.include(request, response);
}
```

The last component in this interaction is the JSP page. The JSP page obtains the name of the action that was predefined to interact with property broker. This value is set in the portlet URL response. When the user presses the submit button, this content is sent to the server through the HTTP request. See Figure 5-5.

**Example 5-5  JSP page for send message portlet**

```html
<%
    PortletURL actionUrl = renderResponse.createActionURL();
    actionUrl.setParameter(SendMessagePortlet.ACTION_NAME_PARAM, 
      SendMessagePortlet.SEND_MESSAGE_ACTION);
%>

<form method="POST" action="<%= actionUrl.toString() %>>"
    <input name="<%=SendMessagePortlet.FORM_TEXT%>" type="text">
    <input name="<%=SendMessagePortlet.FORM_SUBMIT%>" type="submit" value="Submit">
</form>
```

**Receiving a message in a target portlet component**

In many cases, the target portlet does not require updates to receive a message. However, the processAction() method should be prepared to handle the received action from the property broker. Portlet components normally do not differentiate if the action is generated from a keyboard submit or received from the property broker. Also, the WDSL file for this portlet component must always indicate the
input property definition. Example 5-6 shows the processing of an action in the
target portlet.

*Example 5-6  Processing an action in the target portlet*

```java
public void processAction(ActionRequest request, ActionResponse response) throws PortletException, java.io.IOException {

    String actionName = request.getParameter(ACTION_NAME_PARAM);

    if(RECEIVE_MESSAGE_ACTION.equals(actionName)) {
        messageWired = request.getParameter(FORM_TEXT);
    }
}
```

When the portlet receives a request, the processAction() method is triggered and
the code verifies if the action should be processed. In the sample scenario, the
constant RECEIVE_MESSAGE_ACTION represents the action name specified in the
WSDL file.

Table 5-1 shows the relations between all parameters.

*Table 5-1  Sample scenario component locations and values*

<table>
<thead>
<tr>
<th>Component</th>
<th>Location</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>SendMessage Portlet</td>
<td>public String</td>
<td>MessageWireAction</td>
</tr>
<tr>
<td></td>
<td>SEND_MESSAGE_ACTION</td>
<td></td>
</tr>
<tr>
<td>SendMessage WSDL</td>
<td>&lt;binding &lt;portlet:action name</td>
<td>MessageWireAction</td>
</tr>
<tr>
<td>Plugin.xml</td>
<td>extension id= “com.ibm.rcp.portlet.wire”</td>
<td>targetname= “MessageWireAction”</td>
</tr>
<tr>
<td>ReceiveMessage WSDL</td>
<td>&lt;binding &lt;portlet:action name</td>
<td>MessageWireAction</td>
</tr>
<tr>
<td>ReceiveMessage Portlet</td>
<td>public String</td>
<td>MessageWireAction</td>
</tr>
<tr>
<td></td>
<td>SEND_MESSAGE_ACTION</td>
<td></td>
</tr>
</tbody>
</table>

In this sample scenario, once the parameter is processed, the final step is to
pass the message to the JSP for display.

**Defining the WSDL**

If you want to wire a source portlet, you must register its output properties in the
associated WSDL file. Additionally, target portlets associate their supported
actions with an input property declared as an XML type. These actions are
declared in a WSDL file, with a custom binding extension that specifies a mapping from the abstract action declaration to the actual action implementation.

Associated with each action, there is a single input parameter described by an XML type and zero or more output parameters, each described by an XML type. Each input or output parameter encapsulates exactly one property. The input property type is used for matching the action to sources, and its value is filled in when the user triggers the action using a wire.

The output parameters, if specified, are used to automatically trigger other compatible actions (consuming the same type) on other wired portlets every time the action executes and this may also be used to trigger chains of related actions.

The location of the WSDL file is configured as a portlet parameter in the portlet descriptor (portlet.xml) as shown in Example 5-7.

Example 5-7  WSDL location in portlet descriptor (portlet.xml) file

```xml
<portlet-preferences>
  <preference>
    <name>com.ibm.portal.propertybroker.wsdllocation</name>
    <value>/wsdl/SendMessage.wsdl</value>
  </preference>
</portlet-preferences>
```

You must declare the exchange capabilities of this portlet using a WSDL file. The complete WSDL file for the source portlet is provided for you as additional material in this book. For details see Appendix B, “Additional material” on page 721.

For example, follow these steps to create the WSDL file:

1. From the Project Explorer view, right-click the DemoWire/WebContent folder and select New → Folder.
2. Enter wsdl in the folder name field and click Finish.
3. Right-click the wsdl folder just created, choose File → New → Other → WebService → WSDL, and fill in the wsdl properties as shown in Figure 5-7.

Figure 5-7  Using the WSDL editor
Example 5-8 illustrates a sample WSDL for a source portlet when publishing a message to the property broker.

**Example 5-8  Sample WSDL file in a source portlet**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="OrderDetail_Service" targetNamespace="http://www.ibm.com/wps/c2a"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
    xmlns:tns="http://www.ibm.com/wps/c2a"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:portlet="http://www.ibm.com/wps/c2a">

    <message name="WireResponse">
        <part name="wire_wsdl_text" type="portlet:WireStringType"/>
    </message>

    <portType name="Wire_Service">
        <operation name="WireOperation">
            <output message="tns:WireResponse"/>
        </operation>
    </portType>

    <binding name="SendMessageBinding" type="portlet:Wire_Service">
        <portlet:binding/>
        <operation name="WireOperation">
            <portlet:action name="MessageWireAction" type="standard"
                actionNameParameter="ACTION_NAME"/>

            <output>
                <portlet:param name="wire_text" partname="wire_wsdl_text"
                    boundTo="request-attribute"/>
            </output>
        </operation>
    </binding>
</definitions>
```
Target portlets declare their properties in a WSDL file. The process to define the WSDL location and create WSDL definitions are similar. However, the WSDL files for source and target portlets are independent of each other.

Example 5-9 illustrates a sample WSDL file for a portlet receiving a message (target portlet). You can define the list of objects that will be participating in the property broker action. You define these objects with the xsd:simpleType tag.

Example 5-9  Sample WSDL file for a target portlet

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="OrderDetail_Service"
targetNamespace="http://www.ibm.com/wps/c2a"
xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
xmlns:tns="http://www.ibm.com/wps/c2a"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:portlet="http://www.ibm.com/wps/c2a">
<types>
<xsd:schema targetNamespace="http://www.ibm.com/wps/c2a">
<xsd:simpleType name="WireReceivedStringType">
<xsd:restriction base="xsd:string">
</xsd:restriction>
</xsd:simpleType>
</xsd:schema>
</types>
<message name="WireRequest">
<part name="wire_wsdl_text" type="portlet:WireReceivedStringType"/>
</message>
<portType name="Wire_Service">
<operation name="WireOperation">
<input message="tns:WireRequest"/>
</operation>
</portType>
<bringing name="ReceiveMessageBinding" type="portlet:Wire_Service">
<portlet:binding/>
<operation name="WireOperation">
<portlet:action name="MessageWireAction" type="standard"
The plugin.xml file

This file contains information related to Expeditor application properties. Lotus Expeditor uses this file to control the message flow between communicating portlets. The plugin.xml file is used to enable inter-portlet communication or portlet-to-eclipse communication as it defines the wires between source and target components.

In addition, as in the non-portal administered application environment, wires can also be defined and contributed to the Lotus Expeditor platform when using the portal-managed environment. For details see 5.4.2, “Wires declared through WebSphere Portal” on page 112.

The sample scenario shown in this chapter relates to the non-portal administered environment where wires are defined and contributed using the property broker wire extension point com.ibm.rcp.propertybroker.PropertyBrokerWire in the plugin.xml file.

Note: Registering wire definitions with the property broker can be done more efficiently using the RAD plug-in editor.

For example, follow these steps to register wires in the non-portal environment:

1. Right-click the plugin.xml descriptor file and select Open With → Plug-in Manifest Editor.
2. Select the **Extensions tab** as shown in Figure 5-8.

![Figure 5-8  Plug-in extensions](image)

3. Select **Add**.

![Figure 5-9  Sample plug-in extension menu](image)

4. In the New Extension window, deselect **show only extension points from the required plug-ins** to view all available plug-ins and type `com.ibm.rcp.propertybroker.PropertyBrokerWire`.
5. Click Finish.

6. Next, insert the properties for this plug-in. For example:
   
   a. **type** is type of the wire. The type must be `PROPERTY_TO_ACTION` for inter-portlet communication or portlet-to-eclipse communication.
   
   b. **sourceentityid** is the name of the wire source. If the source is a JSR 168 portlet, the value of this field is the Uniform Resource Identifier (URI) of the source portlet window. In this example, the value is `/.DemoWire/SendMessage/default`. The source portlet window information can be found in the plugin.xml file in two different tags:
      
      i. `<contextRoot>`: this tag has a context to call all portlets on this portal application
      
      ii. `<portlet-name>`: it has a name of the portlet that you want to call
   
   c. **targetentityid** is the name of the target. If the target is a JSR 168 portlet (basic portlet), the value of this field is the URI of the target portlet window. The URI is comprised of the context root, portlet name, and portlet window name. The process is the same for sourceentityid but in this case it is for the target. In this example the value is `/.DemoWire/ReceiveMessage/default`.
   
   d. **targetname** is the name of the target action. The value must come from the WSDL of the target portlet, and it is the name attribute of the `<portlet:action>` element. In this sample the name is `MessageWireAction`. If the target is an Eclipse component, the value of this field is the Eclipse view ID.
   
   e. **sourceparam** is the name of the source parameter. The value must come from the WSDL of the source portlet or Eclipse component, and it is the name of the `<portlet:param>` element in the output section.
   
   f. **targetparam** is the name of the target parameter. The value must come from the WSDL of the target portlet or the Eclipse component, and it is the name of the `<portlet:param>` element in the input section.

   **Note:** If the source is an Eclipse component, the value of this field is the Eclipse view ID. The URI is comprised of the context root, portlet name, and portlet window name.

---

**Example 5-10  Declaring a wire in the plugin.xml file**

```xml
<extension id="com.ibm.rcp.portlet.wire" name="Portlet Wire"
   point="com.ibm.rcp.propertybroker.PropertyBrokerWire">

   <wire ordinal=""/>
```
5.3.2 Rich Client Platform components

RCP components, as compared to portlets, do not have generic actions that can be used by the property broker to call the component. They also do not have a central action for processing the business logic in an Model View Control (MVC) like way. In an RCP component, a view or dialog might have numerous buttons or other widgets that can cause events to happen. They all can trigger the change of properties and thus, the transmission of data across the property broker.

Receiving data in RCP components

The property broker uses existing Eclipse or Java interfaces to call into a component. For components, which do not want to implement Standard Widget Toolkit (SWT)/JFace user interface elements, the least costliest way to implement listening to property broker events is to create a handler class that implements the org.eclipse.core.command.IHandler interface. The method, which is called by the property broker, is execute (ExecutionEvent event).

The PropertyChangeEvent is set as the event trigger and can be accessed by calling ExecutionEvent.getTrigger(). The sample code shown in Example 5-11 illustrates how a property broker change event in an Eclipse IHandler action is processed.

Example 5-11 Receiving an event in an IHandler handler

```java
public Object execute(ExecutionEvent event) throws ExecutionException {
    if (event.getTrigger() instanceof PropertyChangeEvent) {
        final Display display = Display.getDefault();
        final PropertyChangeEvent pce =
            (PropertyChangeEvent) event.getTrigger();
        ...
    }
}
```
Richer components can provide a handler that either implements the org.eclipse.jface.action.IAction interface or extends the org.eclipse.jface.action.Action class. The event passed into this method is of type Event (org.eclipse.swt.widgets.Event) and implements the PropertyChangeEvent (com.ibm.rcp.propertybroker.event.PropertyChangeEvent) interface. Unlike the IHandler implementation, the SWT event that passed in can simply be typecast to the PropertyChangeEvent interface. The sample code shown in Example 5-12 illustrates the broker's event interface in an SWT IAction.

Example 5-12  Receiving an event in an IAction handler

```java
public void runWithEvent(Event event) {
    //Make sure the Event is from the Property Broker
    if (event instanceof PropertyChangeEvent) {
        final Display display = Display.getDefault();
        final PropertyChangeEvent finalEvent =
                        (PropertyChangeEvent)event;
        ...
    }
}
```

Registering the handler with the property broker

Providing an entry point for the property broker to call the component is the first task that has to be performed. The second is to register the handler with the property broker. This is accomplished by using an Eclipse extension point that the Lotus Expeditor platform implements. Its name is com.ibm.rcp.propertybroker.PropertyBrokerDefinitions and it can declare a handler that defines a class name, a WSDL file location and an action type. Section 5.2, “Declaring data items and actions” on page 87 has already explained the purpose of the WSDL file and its contents.

The action type can be SWT_ACTION, COMMAND or PORTLET. For RCP components, you must use SWT_ACTION.

Example 5-13  plugin.xml excerpt showing a property broker definition

```xml
<extension point="com.ibm.rcp.propertybroker.PropertyBrokerDefinitions">
    <handler class="com.ibm.itso.compapp.carrental.creditcard.CreditCardActionHandler" file="CreditCardComponent.wsdl" type="SWT_ACTION"/>
</extension>
```
If you have created a proper WSDL file, implemented a handler class and declared the handler properly in your plug-in’s plugin.xml file, your component is ready to receive data from the property broker.

**Sending data from an RCP component**

While some coding and declarations must be performed to receive data from the property broker, sending data is easier. In the simplest case, it does not even need a WSDL file to allow transmission of property change events. However, without a proper WSDL file, users are not informed about what you are able to provide and how you provide it. Someone cannot wire your component using one of the existing wiring tools referred to in 5.4, “Declaring wires” on page 109, nor would they know how to receive your data even without a wire.

**Note:** You should always provide a WSDL file in components that participate in communication.

**Preparing for the communication**

Communication through the property broker is usually triggered by the fact that a property’s value has changed. There is only one exception to this, which is the C2A scenario, which will be explained later in this section.

To inform the property broker about a changed value of a property, you have to prepare the parameters that you can pass to the property broker for it to process them and inform subscribed components about the change.

The first thing to do is to describe the property. A complete property description must contain the following:

- The namespace where the property is declared in
- The data type the property is using
- The name of the property

These three values must uniquely identify a property throughout the system. Once the description of the property is complete, a value gets assigned to this description. The whole of this can then be passed into the property broker for further processing.

Example 5-14 illustrates how to create a property and assign a value to it.
Example 5-14  Preparing property values

```java
... 
PropertyController prop = PropertyFactory.createProperty();
prop.setNamespace(BookingConstants.PROPERTY_NAMESPACE);
prop.setName(BookingConstants.PROPERTY_CAR_SIZE_INPUT);
prop.setType(BookingConstants.PROPERTY_TYPE);
PropertyValue value = PropertyFactory.createPropertyValue(prop,
customer.getPref());
... 
```

At first, an empty property is created through the PropertyFactory. It is then provided with the three data items that uniquely identify the property. In this case we used constants that are defined in a public class making it easier to rename property descriptions. The last line of the example shows how to create the actual property value. Again a factory function is used to pass into the description of the property and its value, and a ready made PropertyValue instance is returned by this.

**The proper wire way**

Using wires and how they are declared is explained in 5.4, “Declaring wires”. This is the safest way to communicate. First, it makes sure that no other component receives your data accidentally, and in the worst case, this could even pose a security hole. Second, it obeys the choices of the application assembler: the person that took various components, arranged them to form an application and asked for certain communication between them. In addition to this, wired communication exhibits the advantage of communicating multiple property change events with just one call. Another reason why you should use wired communication is portlets can only participate in wired communication. If you used the broadcast way, which is explained below, your property changes would not reach any portlets.

To prepare for a wired communication, you have to create PropertyValue instances for all the property changes you want to communicate. Regardless of whether you communicate just one property or multiple properties, you have to collect your PropertyValue instances in an array. This array can then be passed to the property broker.

There is one additional parameter that must be provided to the property broker to make it identify the proper wire. Each wire has a so-called source entity ID and a target entity ID. In order for the property broker to identify a wire, you must provide the source entity ID with your property change event.

For Lotus Expeditor, entity IDs are basically the view IDs that are used by Eclipse. However, as a view can be used in multiple places, sometimes even on
the same page, a secondary ID is required to uniquely identify a view. If you declared your application through a perspective as described in 14.9, “Laying out applications programmatically” on page 367, you might not have provided any secondary IDs for your views. You must, however, always do so. Property broker assumes, that all views have secondary IDs.

Example 5-15 shows how a value array is passed to property broker for it to fire the change events across the wires that have been drawn for this particular source entity.

Example 5-15  Communication property changes using a wire

```java
PropertyValue[] values = new PropertyValue[] { value };

SWTHelper.changedProperties(values, view);
```

As you can see, the view is passed into the changedProperties function. To allow the SWTHelper function to make up the source entity ID, it needs to know which view is the source of this property change. SWTHelper.changedProperties is basically (but not necessarily) a substitute for this. Example 5-16 shows potential coding for the SWTHelper.changedProperties method.

Example 5-16  Potential coding for the SWTHelper.changedProperties method

```java
PropertyBroker p = PropertyBrokerFactory.getBroker();

String primid = view.getViewSite().getId();
String secid = view.getViewSite().getSecondaryId();
String fullid = primid;

if (secid != null && secid.compareTo("") != 0)
    fullid += "." + secid;

p.changedProperties(values, fullid);
```

Ultimately, it all comes to the changedProperties call of the actual broker. Now property broker takes over, identifies all existing wires based on all the properties that you passed in, and subsequently calls all actions of the receiving components to retrieve the changed values.
The broadcast way

If, for whatever reason, you do not want to use a wire, or you cannot use wires, you can have property broker informing other components about your property changes as long as the type and namespace of the properties match. As pointed out earlier, this is a potential security issue, because any arbitrary component can listen to your property changes, take away the data and do something with it. If your data is a color label for a shirt, that is not an issue, but it is a more serious matter if it is a client’s credit card number.

The preparation for a broadcast is the same as for a wired communication, the property needs to be described and a PropertyValue instance has to be created. This time, however, you cannot take an array to communicate multiple changes in one go, instead you have to retrieve all potential actions for one particular property from the property broker, after which you tell it to call those actions. Example 5-17 shows you how to do that.

Example 5-17  Sending a property change as a broadcast

```java
PropertyBroker broker = PropertyBrokerFactory.getBroker();
Action actions[] = broker.getActionsFromMatchingProperty(prop);
broker.callActions(actions, value, "", ");
```

After having set up the property instance that describes our property, the property broker is asked to find all components that are interested in receiving such a property and to return an array with those actions. In the next stop, the property broker calls all the actions in this actions’ array. Potentially, you could have parsed that array and removed some of the actions on whatever algorithm is suitable for this.

Note: A component cannot retrieve multiple values in one call of its action. Your component needs to be prepared to retrieve item by item and only proceed with further processing once all the required data items have been received.

Note: As pointed out earlier, your broadcast will not be received by portlets that have expressed interest in this particular property. Portlets can only receive change events if they have been wired to them.

The Click-to-Action way

C2A is a technology that was introduced to WebSphere Portal by IBM with the advent of WebSphere Portal 5.0. It provides a way for a user to click on a data
item (property) and be presented with a choice of potential actions. The user can then select such an action and WebSphere Portal transfers the property to the selected action. One could call this an “on-the-fly-wire on the user’s grace”.

This way of communication is also provided by a SWTHelper method. The method’s name is showAvailableActions. It takes an SWT Control, an object owner, a target view and a property value as its input and automatically presents the user with a context menu, allowing the user to choose from the gathered actions. If more than one action is available, the menu even appends an “Execute all actions” entry at the end of the menu.

Example 5-18 illustrates a Right-Mouse-Button-Context-Menu implementation for an SWT table viewer control.

```java
Example 5-18   Right mouse button context menu implementation

public void mouseUp(MouseEvent arg0) {
    if (arg0.button == 3) {
        String selection = viewer.getSelection().toString();
        PropertyController prop = PropertyFactory.createProperty();
        prop.setNamespace(Constants.PROPERTY_C2A_NAMESPACE);
        prop.setName(Constants.PROPERTY_C2A_NAME);
        prop.setType(Constants.PROPERTY_C2A_TYPE);
        PropertyValue value =
            PropertyFactory.createPropertyValue(prop, selection);

        SWTHelper.showAvailableActions(
            viewer.getControl(),
            _this,
            null,
            value);
    }
}
```

As you can see, the target entity view is passed as a null. This indicates to the property broker that it should not reduce the choice of potential actions to a particular target entity. This resembles the broadcast type of communication, which means that the property broker will not be able to identify wires for this communication, and hence, portlets will be left out.

5.4 Declaring wires

Wires provide a way for controlling who is talking to whom. You can perform component to component communication without wiring, using a kind of
broadcast mechanism as described in “Sending data from an RCP component” on page 105, but that can have numerous side effects, especially if a component is used in multiple places.

To draw a line, you have to know where the line starts and where it ends. To put a wire between two properties, you will also have to know the exact docking points of the wire-ends. A unique representation of such a docking point is made up by:

- The property’s namespace
- The property’s name
- The property’s type
- The component’s entity ID

If you take those four components, you can uniquely identify any property in the system. As described in 5.2, “Declaring data items and actions” on page 87, the first three parts for this identification come from the WSDL file, which declares the properties and actions for a certain component.

The entity ID has also been discussed previously, it is the combination of the primary and secondary ID of an Eclipse view. Equipped with this data, you can now add wires between your components, and there are several ways to do so. Here we show you how to add wires to the plugin.xml file and how to do this using the WebSphere Portal tools.

### 5.4.1 Wires as part of your plugin.xml

As with the declaration of property broker information in your plugin.xml described in “Registering the handler with the property broker” on page 104, there is a similar way to register a wire within your plug-in’s plugin.xml file. The extension point that has to be used is called com.ibm.rcp.propertybroker.PropertyBrokerWire.

The important part of it is the wire tag which accepts as parameters. See Table 5-2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>(optional) The name of the wire.</td>
</tr>
<tr>
<td>title</td>
<td>(optional) The title of the wire. Has no impact on the wire.</td>
</tr>
<tr>
<td>ownerid</td>
<td>(optional) Owner ID. With this ID you can then query the property broker from the APIs with this string. Otherwise, the owner ID will be populated by the property broker.</td>
</tr>
</tbody>
</table>
A practical example of a plugin.xml-based wire definition is described in Example 5-19. It wires the booking components' property CCNumber to the credit card component's property Credit Card Number by calling the receiveAuthAction of the credit card component.

**Example 5-19  Example plugin.xml based wire**

```xml
<extension
    id="com.ibm.itso.compapp.carrental.creditcard.CreditCardNumberWire"
    name="Authorization Wire"
    point="com.ibm.rcp.propertybroker.PropertyBrokerWire">
    <wire enable="true" ordinal="100"
        sourceentityid=
            "com.ibm.itso.compapp.carrental.booking.BookingView:booking.secondary.id"
        sourcename="BK Credit Card Number"
        targetentityid=
            "com.ibm.itso.compapp.carrental.creditcard.CreditCardView:creditcard.secondary.id"
        targetname="receiveAuthAction"
    />
</extension>
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordinal</td>
<td>(optional) The value of this wire and how it is sorted in the broker registry</td>
</tr>
<tr>
<td>type</td>
<td>Currently the only type supported is the PROPERTY_TO_ACTION wire type.</td>
</tr>
<tr>
<td>sourcename</td>
<td>The source property name for this wire. This value is defined as the name</td>
</tr>
<tr>
<td></td>
<td>value of your portlet:param tag in the output section.</td>
</tr>
<tr>
<td>sourceentityid</td>
<td>This is the source's entity IDa.</td>
</tr>
<tr>
<td>sourceparam</td>
<td>The source parameter that will be included with the wire information when</td>
</tr>
<tr>
<td></td>
<td>sent to the Action.</td>
</tr>
<tr>
<td>targetname</td>
<td>The Action name defined in the WSDL file.</td>
</tr>
<tr>
<td>targetentityid</td>
<td>This is the target's entity IDa.</td>
</tr>
<tr>
<td>targetparam</td>
<td>The target parameter that will be included with the wire information when</td>
</tr>
<tr>
<td></td>
<td>sent to the Action. Portal will put the target's property name into this</td>
</tr>
<tr>
<td></td>
<td>parameter.</td>
</tr>
<tr>
<td>enable</td>
<td>This will have the wire automatically enabled when it is registered.</td>
</tr>
<tr>
<td>uid</td>
<td>(optional) A unique ID for this specific wire. If not supplied, a unique ID</td>
</tr>
<tr>
<td></td>
<td>will be generated internally.</td>
</tr>
</tbody>
</table>

---

a. Entity IDs are usually primary and secondary Eclipse view IDs separated by a colon. If one view represents a portlet, the primary view ID is the portlet's context-root and its name separated by a forward slash.
Declaring wires through a plugin.xml file is an error prone way of linking your components. During the initial load of Lotus Expeditor, it will not complain about non-existing properties, actions or entity IDs. If during run-time, you fire a property change event, property broker will search for an appropriate wire and silently does nothing if one is not found. You can, however, trace what property broker is doing when you change a property. This can help you finding mistakes in your property or wire declarations or both.

Note: A simple misspelling in either of your declarations will render a broken wire.

5.4.2 Wires declared through WebSphere Portal

If your application is managed through WebSphere Portal, you can also define wires by using the wiring tool that comes with WebSphere Portal. In order to make WebSphere Portal aware of your properties and actions, the portlets that you use to lay out your application with the composite application editor must have a parameter. This makes WebSphere Portal know where your WSDL file is located. If your component is a portlet that is supposed to run on Lotus Expeditor, all you have to do is to add the location of your WSDL file into your portlet's portlet.xml file. Example 5-20 on page 114 shows an example including the setting of a WSDL file location.

When placing non-portlet components on a page, you can use instances of the Rich Client View Placeholder portlet to place your components, which is described in 14.5.1, “Portlet instances versus portlet clones” on page 354. This is going to change if WSDL files are involved. Because you cannot add your WSDL file to the existing Rich Client View Placeholder portlet, you have to provide your own portlet in your own portlet application's WAR file.

Fortunately, WebSphere Portal does not add many requirements on these portlets. If you are using RAD to develop your components, use RAD's portlet project creation wizard to have it create the whole portlet for you, the most basic portlet will do. You do not have to change any of the code that was generated. All you need to do is to place your WSDL file somewhere below the WebContent folder of your newly created portlet project and specify this location in the portlet.xml file. Usually WSDL files are contained inside a folder called wsdl. It is a good idea to also create a folder called wsdl inside the WebContent folder and put the file in there.
Figure 5-10 shows you what your project will look like.

Figure 5-10  Content of a portlet project with a WSDL file

Now open your portlet's portlet.xml file, which resides in the WEB-INF folder inside the WebContent folder, and add the portlet preference. Using the Portlet Deployment Descriptor editor, Figure 5-11 shows what it would look like.

Figure 5-11  Portlet Deployment Descriptor editor
As an alternative, if you do not have RAD and its editor, you can manually add the lines, shown in Example 5-20, to your portlet.xml.

Example 5-20  Excerpt of portlet.xml to declare a WSDL file location

```xml
...
  <portlet-preferences>
    <preference>
      <name>com.ibm.portal.propertybroker.wsdllocation</name>
      <value>/wsdl/CreditCardComponent.wsdl</value>
    </preference>
  </portlet-preferences>
...
```
Portal composite applications

This chapter describes how portlet components can be prepared for communication in a portal environment. We provide the necessary information and steps to enable and perform inter-portlet communication between portlet components. Specifically, in this chapter you will find information to:

- Create WSDL files defining the bits and pieces of component intercommunication
- Update your code to allow component communication to occur
- Create *wires* between components to control communication
6.1 Overview

A significant advantage of portlet components is the ability for the components to communicate with each other and thus enabling the components in the portlet application to be interactive. Composite applications are generally referred to as cooperative portlets when used in the context of inter-portlet communications. The main idea between composite applications and cooperative portlets is the enablement of multiple components to communicate over the same medium to exchange data during runtime.

- The communication between the two components has been developed into the component. In essence, the components knew of each other at the time they were developed.

- A neutral method in which to enable communications is done by separating the declarations of data items (that is, the data you want to provide or receive) and the communications layer (that is, the medium of communication) from the component code.

The preferred method is the latter case. However, in order to be truly neutral between components, the declaration of the data items has to be specific enough to guarantee proper transfer of data while being flexible enough to allow linking of components using generic tooling. Both issues have been achieved using a Web Services Definition Language (WSDL) file where the data is described.

Composite (cooperative) portlets are available in WebSphere Portal 5.1 and later. Based on design, one or more target portlets on a portal page can automatically react to changes from a source portlet when an event is received. Portlets that are targets of these events can react so that users are not required to make repetitive changes or actions in other portlets on the page thus providing a coordinated and consistent behavior between the portlets on the page while enhancing user experience. Cooperation between source and target portlets is facilitated by a WebSphere Portal runtime entity called the PropertyBroker. Portlets on a page can cooperate in this way even if they were developed independently, without awareness of other cooperative portlets.

6.2 Portlet component scenario

In this scenario, you will develop cooperative portlet components for the ITSO Car Rental sample application that will run within the WebSphere Portal environment. These portlet components will integrate, in the chapters to follow, into existing components of the ITSO Car Rental sample composite application. This scenario covers development, assembly, and deployment of these “wired”
components to WebSphere Portal and consists of two views, a user view and an administrator's view.

The cooperative portlet components communicate with each other via the property broker and will be developed using the JSR-168 portlet standard. Once, these components are developed and deployed to the WebSphere Portal environment, the general flow of this application from the user point of view will be executed as follows:

1. The user enters a customer ID to be published to the property broker.
2. Receive the customer ID from the property broker and generate a list of past reservations. When a one of these reservations is selected, the reservation ID is published to the property broker.
3. The reservation ID is received from the property broker to show reservation details.

The three portlet components in this sample composite application are illustrated in Figure 6-1.

The execution flow of the application development you will perform for the user view is outlined below:

1. Portal invokes the sample composite application with the three portlet components as shown in Figure 6-1.
2. When the user submits the request with their customer ID, the processAction() method in the Customer ID portlet identifies if the action and value submitted should be published to the PropertyBroker to put onto the "wire". The output property must be defined in the UserReservationsProject portlet's WSDL file.
3. The PropertyBroker obtains the parameter and identifies the target components.
4. The processAction() method in the Receive Customer ID portlet will be invoked and will receive the input property from the PropertyBroker. When a customer number is submitted, the customer’s reservations appear in the Reservations portlet. This portlet, in addition to having an input property defined in the WSDL file, also has an output property defined.

5. When the user invokes an action from one of the reservations, this portlet’s processAction() method interrogates the action submitted and publishes the action value to the PropertyBroker.

6. The PropertyBroker obtains the parameter and identifies the target components.

7. The processAction() method in the Reservation Details portlet will be invoked and will receive the input property from the PropertyBroker and performs additional application function to generate the view as shown in Figure 6-1.

### 6.2.1 Develop portlet components

The instructions provided below take you through the steps necessary to quickly and easily create cooperative portlet applications. In the steps to follow, note that a significant portion of the programmatic code is written for you by the Rational Application Developer tooling wizards and thus significantly speeds-up development time.

In developing the applications, you will perform the following steps.

1. Create shells of portlet components
2. Enabling the wires
3. Examining the auto-generated WSDL files
4. Setting and getting wire data
5. Customizing the view

**Create shells of portlet components**

To create portlets using Rational Application Developer 7.0, you must have the Portal Tools plug-in installed and the development environment must be properly configured.

1. Start Rational Application Developer. **Start → Run → Programs → IBM Software Development Platform → IBM Rational Application Developer**

2. For purposes of this development project, we are using **c:\workspace** as the workspace folder. If prompted, select the workspace location you will use and click **OK**.
3. If not already open, switch to the J2EE perspective by clicking **Window → Open Perspective → Other → J2EE**.

4. Create a new Portlet Project by selecting **File → New → Project → Portlet Project** and click Next. If you are prompted enable Portal Development, click **OK**.

5. In the New Portlet Project window, enter the following values. If a parameter value is not provided below, keep the default setting.
   a. Project name: **UserReservationsProject**
   b. EAR membership: **Uncheck Add project to an ear**
   c. Portlet API: **JSR 168 Portlet**
   d. Create a portlet: **Uncheck this check box**
   e. Click **Finish**.
   f. Notice the output in the Project Explorer as shown in Figure 6-2.

6. If you are prompted to open the Web Perspective, click **Yes**.

7. Next, create the portlet shell for the CustomerIdentification portlet. If a parameter value is not provided below, keep the default setting. Right-click **UserReservationsProject → New → Portlet**.
   a. In the Portlet pane, enter Portlet name of **CustomerIdentification** and **Portlet type** as Basic Portlet. Click **Next**.
   b. Accept the default settings on the Portlet Setting pane and click **Next**.
   c. Unselect all options on the **Action and Preferences** pane and click **Finish**.
8. Repeat the prior step for two additional portlets. Substitute the Portlet name with the following two portlet names.
   a. **CustomerReservations**
   b. **Reservation Details**

9. Your Project Explorer pane should now contain all three portlets as shown in Figure 6-3.

![Project Explorer view](image)

**Figure 6-3  Project Explorer view**

### Enabling the wires

Now that the basic JSR-168 portlets are created, you will create the WSDL files that creates the “wiring” capability. Note that the wires themselves are not “attached” as this is a function handled by WebSphere Portal and is covered in the upcoming sections of this chapter.

1. To begin, expand the Portlet deployment descriptor section and right-click the **CustomerIdentification**, and select **Cooperative → Enable this Portlet to Send Data (source)** as illustrated in Figure 6-4.

![View of portlet descriptor menu options](image)

**Figure 6-4  View of portlet descriptor menu options**
2. From the wizard, enter the following details and leave other entries to their default settings.
   a. Data Type URI: \texttt{http://userreservationsproject\#CustID}
   b. Java Type: \texttt{java.lang.String}
   c. Action Parameter: \texttt{ACTION\_NAME\_PARAM}
   d. Action Value: \texttt{CustIDAction}
   e. Location: Request attribute
   f. Property Name: \texttt{outputCustID}

3. Next, enable the \texttt{CustomerReservations} portlet to receive data sent by the CustomerIdentification portlet. Enter the following details and leave other entries to their default settings.
   a. Data Type URI: \texttt{http://userreservationsproject\#CustID}
   b. Java Type: \texttt{java.lang.String}
   c. Action Parameter: \texttt{ACTION\_NAME\_PARAM}
   d. Action Value: \texttt{CustIDAction}
   e. Location: Request parameter
   f. Property Name: \texttt{inputCustID}

4. In addition to CustomerReservations being a consumer of information, it also acts as a publisher of data. CustomerReservations publishes the reservation ID to the ReservationDetails portlet. Enable the \texttt{CustomerReservations} portlet to send data. Enter the following details and leave other entries to their default settings.
   a. Data Type URI: \texttt{http://userreservationsproject\#ResID}
   b. Java Type: \texttt{java.lang.String}
   c. Action Parameter: \texttt{ACTION\_NAME\_PARAM}
   d. Action Value: \texttt{ResIDAction}
   e. Location: Request attribute
   f. Property Name: \texttt{outputResID}

5. Lastly, enable the \texttt{ReservationDetails} portlet to receive data sent by the CustomerIdentification portlet. Enter the following details and leave other entries to their default settings.
   a. Data Type URI: \texttt{http://userreservationsproject\#ResID}
   b. Java Type: \texttt{java.lang.String}
   c. Action Parameter: \texttt{ACTION\_NAME\_PARAM}
d. **Action Value**: `ResIDAction`  

e. **Location**: `Request parameter`  

f. **Property Name**: `inputResID`  

6. As a result of performing the above steps, three WSDL files have been generated by Rational Application Developers built-in tooling. These files have been placed into the `/WebContent/wsdl` folder of the current project. No further modification to the WSDL files is necessary.  

7. Your output should appear as shown in Figure 6-5.

![Figure 6-5](image)

**Figure 6-5  Project Explorer view of sources and targets of the three portlets**

**Examining the auto-generated WSDL files**

In order to be truly neutral between components, declarations and actions have to be specific enough to guarantee proper transfer of data while remaining neutral to the transport layer. On the other hand, it also needs to be flexible enough to allow linking of components using a generic tool. Both issues have been addressed using Web Services Definition Language (WSDL) files. Based on the prior steps you have completed, three WSDL files were generated:

- `CustomerIdentification.wsdl`
- `CustomerReservations.wsdl`
- `ReservationDetails.wsdl`

Let us examine these files in closer detail to interpret the contents of the WSDL file.

**CustomerIdentification.wsdl**

In `CustomerIdentification.wsdl`, the following has been accomplished:

- Defined a data type called `CustID` of type string in the namespace of `http://userreservationsproject` which has restrictions of `xsd.string`.  

---

122  Building Composite Applications
- Defined a message called CustIDAction_Response that has one data item, CustID_Output, of type xsd.string.
- Defined a port type name CustomerIdentification_Service that has an operation named CustomerIdentification and contains an output message of type CustIDAction_Response.
- Declared a binding called CustomerIdentification_Binding of port type CustomerIdentification_Service containing an operation of type CustomerIdentification. Here, an action is declared with name CustIDAction. For basic portlets, its type is standard. The action name parameter is ACTION_NAME_PARAM.

Example 6-1  CustomerIdentification.wsdl located project’s /wsdl folder

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:portlet="http://www.ibm.com/wps/c2a"
xmlns:tns0="http://userreservationsproject" name="CustomerIdentification_Service"
targetNamespace="http://userreservationsproject.CustomerIdentification/">
<types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://userreservationsproject">
    <xsd:simpleType name="CustID">
      <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
  </xsd:schema>
</types>
<message name="CustIDAction_Response">
  <part name="CustID_Output" type="tns0:CustID"/>
</message>
<portType name="CustomerIdentification_Service">
  <operation name="CustomerIdentification">
    <output message="tns:CustIDAction_Response"/>
  </operation>
</portType>
<binding name="CustomerIdentification_Binding"
type="tns:CustomerIdentification_Service">
  <portlet:binding/>
  <operation name="CustomerIdentification">
    <portlet:action actionNameParameter="ACTION_NAME_PARAM"
caption="Action.Caption2" name="CustIDAction" type="standard"/>
    <output>
      <portlet:param boundTo="request-attribute" caption="Property.Caption2"
name="outputCustID" partname="CustID_Output"/>
    </output>
  </operation>
</binding>
</definitions>
```
CustomerReservations.wsdl

In CustomerReservations.wsdl, the following has been accomplished:

- Defined a data type called CustID and ResID of type string in the namespace of http://userreservationsproject where both CustID and ResID have restrictions of xsd.string.
- Defined a message, CustID_Request that has one data item, CustID_Input, of type xsd.string.
- Defined a message, ResIDAction_Response that has one data item, ResID_Output, of type xsd.string.
- Defined a port type name CustomerReservations_Service that has an operation named CustomerReservations and contains an input message, CustID_Request, and an output message, ResIDAction_Response.
- Declared a binding called CustomerReservations_Binding of port type CustomerReservations_Service containing an operation of type CustomerReservations. Here, an action is declared with name CustIDAction. For basic portlets, its type is standard. The action name parameter is ACTION_NAME_PARAM.

Example 6-2  CustomerReservations.wsdl located project's /wsdl folder

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:portlet="http://www.ibm.com/wps/c2a"
xmlns:tns0="http://userreservationsproject" name="CustomerReservations_Service"
targetNamespace="http://userreservationsproject.CustomerReservations/">
  <types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://userreservationsproject">
      <xsd:simpleType name="CustID">
        <xsd:restriction base="xsd:string"/>
      </xsd:simpleType>
      <xsd:simpleType name="ResID">
        <xsd:restriction base="xsd:string"/>
      </xsd:simpleType>
    </xsd:schema>
  </types>
  <message name="CustID_Request">
    <part name="CustID_Input" type="tns0:CustID"/>
  </message>
</definitions>
```
ReservationDetails.wsdl

In ReservationDetails.wsdl, the following has been accomplished:

- Defined a data type called ResID of type string in the namespace of http://userreservationsproject where ResID has a restriction of xsd.string.
- Defined a message, ResID_Request that has one data item, ResID_Input, of type xsd.string.
- Defined a port type name ReservationDetails_Service that has an operation named ReservationDetails and contains an input message, ResID_Request.
Declared a binding called ReservationDetails_Binding of port type ReservationDetails_Service containing an operation of type ReservationDetails. Here, an action is declared with name ResIDAction. For basic portlets, its type is standard. The action name parameter is ACTION_NAME PARAM.

Example 6-3  ReservationDetails.wsdl located project's /wsdl folder

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
xmlns:portlet="http://www.ibm.com/wps/c2a"
xmlns:tns="http://userreservationsproject.ReservationDetails/
xmlns:tns0="http://userreservationsproject"
targetNamespace="http://userreservationsproject.ReservationDetails/"
<types>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://userreservationsproject">
    <xsd:simpleType name="ResID">
      <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
  </xsd:schema>
</types>
<message name="ResID_Request">
  <part name="ResID_Input" type="tns0:ResID"/>
</message>
<portType name="ReservationDetails_Service">
  <operation name="ReservationDetails">
    <input message="tns:ResID_Request"/>
  </operation>
</portType>
<binding name="ReservationDetails_Binding" type="tns:ReservationDetails_Service">
<portlet:binding/>
<operation name="ReservationDetails">
  <portlet:action actionNameParameter="ACTION_NAME_PARAM"
caption="Action.Caption" name="ResIDAction" type="standard"/>
  <input>
    <portlet:param boundTo="request-parameter" caption="Property.Caption"
name="inputResID" partname="ResID_Input"/>
  </input>
</operation>
</binding>
</definitions>
```
Importing support files - Database and HTML helper
Perform the following steps to import the helper files to the project.
1. In the Java Resources: src folder, create a new package called
   com.ibm.userreservations.db.
2. Right-click the newly created package and select Import.
3. Expand General → File System → Next and point the From Directory to
   the location where you downloaded the DatabaseOperations.java class.
4. Select the DatabaseOperations.java file and click Finish.

Importing support files - CSS helper
Perform the following steps to import the CSS file to the project.
1. Right-click the WebContent folder and select New → Folder.
2. Right-click the css folder and select Import.
3. Expand General → File System → Next and point the From Directory to
   the location where you downloaded the itso.css stylesheet file.
4. Select the itso.css file and click Finish.

Setting and getting the wire data
Next, you will perform operations necessary to set and get properties from the
PropertyBroker.
1. First, we must add values to the CustomerIdentification portlet to send the
   appropriate value. To do this, a form must be created in the
   CustomerIdentificationPortletView.jsp. Open this file by navigating to
   /WebContent/_CustomerIdentification/jsp/html/.
2. Insert the code shown in Example 6-4 into the file and save the file.

Example 6-4  CustomerIdentificationPortletView.jsp form to submit customer ID

```html
<link rel="stylesheet"
   href='<%= renderResponse.encodeURL(renderRequest.getContextPath() +  "/css/itso.css") %>'
   type="text/css">

<div id="CarSelect">
  <b><font face="Arial" size="2">
   Retreive your reservations
  </font></b>
  <p style="margin-bottom: 3px">
    <font face="Arial" size="2">
      Enter your customer number to begin.<br/>
    </font>
  </p>
</div>
```
3. Next, open the CustomerIdentificationPortlet.java class in com.ibm.userreservationsproject package.

4. Add the code shown in Example 6-5 into the processAction() method. Save and exit the file.

Example 6-5  Code to be added to processAction() in bold

```java
public void processAction(ActionRequest request, ActionResponse response) throws PortletException, java.io.IOException {
    String actionValue = request.getParameter("ACTION_NAME_PARAM");
    if(actionValue != null && "CustIDAction".equals(actionValue)) {
        String formInputCustID = request.getParameter("myCustIDTextbox");
        request.setAttribute("outputCustID", formInputCustID);
    }
}
```

5. So far, we have enabled the portlet to communicate with the PropertyBroker. However, there is currently no logic to handle the logic on the target portlet. Therefore, we have to add code shown in Example 6-6 into the CustomerReservationsPortlet. Open the CustomerReservationsPortlet.java class in the com.ibm.userreservationsproject package and add the following code into the processAction(...) method.

Example 6-6  Code to be added to processAction() in bold

```java
public void processAction(ActionRequest request, ActionResponse response) throws PortletException, java.io.IOException {
    //** This code handles the 'wired' customer ID input
    String actionValue = request.getParameter("ACTION_NAME_PARAM");
    if(actionValue != null && "CustIDAction".equals(actionValue)) {
        custID = request.getParameter("inputCustID");
    }

    //** This code handles the 'wired' reservation ID output
```
else if(actionValue != null && "ResIDAction".equals(actionValue)) {
    String resID = request.getParameter("resID");
    request.setAttribute("outputResID", resID);
}

6. In the same file, CustomerReservationsPortlet.java, add the following line of code to the doView(...) method. Note: add the code prior to the invocation of JSP rendering as shown in as shown in Example 6-7. Save the file.

Example 6-7  Code to be added to doView(...) in bold

```java
public void doView(RenderRequest request, RenderResponse response)
    throws PortletException, IOException {
    // Set the MIME type for the render response
    response.setContentType(request.getResponseContentType());

    //** Add parameter to the request object
    request.setAttribute("custID", custID);

    //** The following code creates the portlet view and is provided to
    //** save you time from writing HTML and database connection code
    //** utilized in this scenario.
    DatabaseOperations dbOps = new DatabaseOperations();
    String reservationsTable = dbOps.getReservations(custID, response);
    request.setAttribute("reservationsTable", reservationsTable);

    // Invoke the JSP to render
    PortletRequestDispatcher rd =
        getPortletContext().getRequestDispatcher(getJspFilePath(request, VIEW_JSP));
    rd.include(request, response);
}
```

7. Once the file has been saved, you will notice an error stating “custID cannot be resolved”. To resolve this, add the following class level String declaration as a class variable as shown in Example 6-8. Save the file and exit.

Example 6-8  Code to be added to CustomerReservationsPortlet.java as class variable

```java
public class CustomerReservationsPortlet extends GenericPortlet {
    public static final String JSP_FOLDER    =
        "/_CustomerReservations/jsp/";    // JSP folder name
    public static final String VIEW_JSP      =
        "CustomerReservationsPortletView";
```
private String custID = new String();
...

8. Next, create a form in the CustomerReservationsPortletView.jsp which will create the reservations listing. Open this file by navigating to /WebContent/_CustomerReservations/jsp/html/.

9. Insert the code shown in Example 6-9 into the file and save the file.

Example 6-9 CustomerReservationsPortletView.jsp form to receive customer ID and send Reservation ID

```html
<link rel="stylesheet" href='<%= renderResponse.encodeURL(renderRequest.getContextPath() + "/css/itso.css")%>' type="text/css">
<div id="CarSelect">

<b><font face="Arial" size="2">Retrieve your reservations</font></b>
<p style="margin-bottom: 3px">
<font face="Arial" size="2">Enter your customer number to begin.</font>
</p>

<form method="POST" action="<portlet:actionURL/>">
    <input name="ACTION_NAME_PARAM" value="CustIDAction" type="hidden"/>
    <input name="myCustIDTextbox" value="" type="text" size="10" maxlength="10" />
    <input name="submit" value="Submit" type="submit" />
</form>

</div>
```

10. Similar to how the CustomerReservationsPortlet.java was modified, modify the ReservationDetailsPortlet.java file as shown below. This will enable the portlet to retrieve and process values from the PropertyBroker. Open ReservationDetailsPortlet.java class in the com.ibm.userreservationsproject package and add the following code into the processAction(...) method also as shown in Example 6-10.

Example 6-10 Code to be added to processAction() in bold

```java
public void processAction(ActionRequest request, ActionResponse response) throws PortletException, java.io.IOException {

    //** This code processes the ‘wired’ reservation ID input
    String actionValue = request.getParameter("ACTION_NAME_PARAM");
    if(actionValue != null && "ResIDAction".equals(actionValue)) {
        resID = request.getParameter("inputResID");

```
if (resID.length() > 0) {
    DatabaseOperations dbOps = new DatabaseOperations();
    reservationDetailsTable = dbOps.getReservationDetails(resID);
    resID = "";
}

11. In the same file, ReservationDetailsPortlet.java, add the following lines of code to the doView(...) method prior to the invocation of JSP rendering as shown in Example 6-11. Save the file.

Example 6-11 Code to be added to doView(...) in bold

```java
public void doView(RenderRequest request, RenderResponse response) throws PortletException, IOException {
    // Set the MIME type for the render response
    response.setContentType(request.getResponseContentType());

    /** Add the parameter to the request object */
    request.setAttribute("resID", resID);

    /** The following code creates the portlet view and is provided to */
    /** save you time from writing HTML and database connection code */
    /** utilized in this scenario. */
    request.setAttribute("reservationDetailsTable", reservationDetailsTable);
    DatabaseOperations dbOps = new DatabaseOperations();
    reservationDetailsTable = dbOps.getReservationDetails(resID);

    // Invoke the JSP to render
    PortletRequestDispatcher rd =
        getPortletContext().getRequestDispatcher(getJspFilePath(request, VIEW_JSP));
    rd.include(request, response);
}
```

12. Once the file has been saved, you will notice an errors stating “resID cannot be resolved”. To resolve this, add the following String declaration as a class variable as shown in Example 6-12. Save the file and exit.

Example 6-12 Code to be added to ReservationDetailsPortlet.java as class variables

```java
public class ReservationDetailsPortlet extends GenericPortlet {
    public static final String JSP_FOLDER = "/_ReservationDetails/jsp/"; // JSP folder name
```
public static final String VIEW_JSP = "ReservationDetailsPortletView";

private String resID = new String();
private String reservationDetailsTable = "";

...

13. Lastly, create the output table for the ReservationDetailsPortletView.jsp which provides the reservation details. Open this file by navigating to /WebContent/_ReservationDetails/jsp/html/.

14. Insert the code shown in Example 6-13 into the file and save the file.

Example 6-13 ReservationDetailsPortletView.jsp form to receive reservation ID

...<link rel="stylesheet"
href='<%= renderResponse.encodeURL(renderRequest.getContextPath() + "/css/itso.css")%>
type="text/css">
<div id="CarSelect">

<b><font face="Arial" size="2">
Select a reservation to obtain additional details.
</font></b>

<%= String reservationsTable = (String)renderRequest.getAttribute("reservationsTable"); %>

<%= reservationsTable %>

6.2.2 Assemble composite application portlet components

In the prior steps, you developed the portlet components that make up the cooperative (or composite), portlet application. The next steps include assembling the components to function in the portal. The assembly process involves administration tasks that join the wires created in the WSDL files which in-turn allow the portlets to communicate with each other.

Export the application WAR file

Complete the following steps to export the WAR file from Rational Application Developer.
1. First, export the UserReservationsProject as a WAR file by right-clicking the UserReservationsProject and selecting Export → WAR file.

2. When prompted to enter the destination, enter a destination where the WAR file is to be saved. In our scenario, we have entered c:\UserReservationsProject.war.

3. Leave other fields to their default setting and click Finish.

**Install the application in WebSphere Portal**

Install the exported application into WebSphere Portal.

1. Open a Web browser and login the WebSphere Portal server using a user ID that has administrative privileges.

2. Install the UserReservationsProject portlet application. Left-click the Launch menu → Administration → Portlet Management → Web Modules → Install.

3. From the Manage Web Modules portlet, left-click Browse and navigate to the location where the UserReservationsProject.war file has been saved. Click Next and then Finish to complete the portlet application installation.

**Assemble the application’s wiring**

Assemble the portlet components and enable communication between the components.

1. Navigate back to the portal Home page.

2. Next, create an empty page. For our scenario, we have created a page named User Reservations under the Getting Started label.

3. Add the three portlets to the User Reservations page and click Done.

4. Refresh the page and the results appear as shown in Figure 6-6.
5. Enter customer number **1234** and click **Submit**. Notice that no actions are occurring although 1234 is a valid customer number. This occurs since WebSphere Portal has not linked the inputs and outputs of the WSDL files.

6. Left-click the **Page menu** of **UserReservations** and select **Edit Page Layout**.

7. Navigate to the Wires page. You will see an information message stating “**EJPAR2050I: No wires have been created on the page.**” as shown in Figure 6-7.

8. First, create the wire that provides the customer number to the **CustomerReservations** portlet. Enter the following settings and then left click the **+** action button, and click **Done**.
a. Source portlet: **UserReservationsProject**
b. Sending: **outputCustID Property**
c. Target page: Default value of **User Reservations**
d. Target portlet: **CustomerReservations**
e. Receiving: **CustIDAction Action,inputCustID Property**
f. Wire Type: **Personal**

9. From the User Reservations page, enter customer number 1234 again and click **Submit**. Notice that the CustomerReservations portlet has displayed a result. This is the result of the message being obtained by the WSDL of the customerReservations portlet and thus invoking the processAction(...) method of the CustomerReservations portlet.

10. Left-click the Reservation ID of “4”. Notice that the ReservationDetails portlet does not receive the message nor take any action. In fact, the ReservationDetails portlet remains with the same view as shown in Figure 6-8. Again, the wire must be created from the Wires page.

![Figure 6-8 Sample of action on CustomerReservations portlet](image)

11. From the User Reservations page menu, select Edit Page Layout, navigate to the Wires page and enter the following settings. When the settings have been entered, left click the + action button, and click **Done**.
   a. Source portlet: **CustomerReservations**
   b. Sending: **outputResID Property**
   c. Target page: Default value of **User Reservations**
   d. Target portlet: **ReservationDetails**
   e. Receiving: **ResIDAction Action,inputResID Property**
   f. Wire Type: **Personal**
When returned back to the User Reservations page, left-click Reservation ID 4 and notice the ReservationDetails portlet takes action and populates data based on the message received by the CustomerReservations portlet.

![Figure 6-9](image)

**Figure 6-9 Sample of action on ReservationDetails portlet**

### 6.3 JavaServer Faces portlet considerations

JavaServer Faces (JSF) is a technology that helps you build user interfaces for dynamic Web applications that run on the server. The JavaServer Faces framework manages UI states across server requests and offers a simple model for the development of server-side events that are activated by the client. JSF is consistent and easy to use. JSF is a standard defined by JSR 127.

In this section we will explain the specifics of enabling cooperative portlet behavior between JSF portlets. To do so we created a portlet project called CustomersPortlet with two portlets:

- **CustomerIdentification source portlet.** This portlet displays a summary information about all customer registered. When you click the customer ID link then the CustomerInformation portlet will send an instance of the com.ibm.itso.customers.beans.CustomerDetail bean to the CustomerDetails portlet.

- **CustomerDetails target portlet.** This portlet displays detail information about the customer ID selected in CustomerIdentification portlet.

Use Portal Wiring Tools to establish the wire between portlets. Once established you can select a customer ID in the CustomerIdentification portlet to send the details of the customer selected to CustomerDetails portlet as shown in Figure 6-10.
6.3.1 Enabling the source portlet

Enabling the source portlet involves the following steps:

- Describe the source properties and actions with WSDL file. The WSDL file contains the following sections:
  - Types: We describe a complex type called CustDetailType that can be emitted by this portlet.
  - Messages: This section describes the CustDetail_Response messages that can be generated by the portlet.
  - Port Type: This section describes the abstract interface of the portlet as seen by the property broker.
  - Binding: An action named sendCustDetailResponse is declared, the type is standard because this is the type use for JSR 168 portlets. The output parameter name is customerInfo, this parameter is bound to request-attribute. You also need to add the class attribute that specifies the fully qualified java type that will pass between portlets, in our case: com.ibm.itso.customers.beans.CustomerDetail.
Example 6-14  CustomerInformation.wsdl file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
 xmlns:portlet="http://www.ibm.com/wps/c2a"
 xmlns:tns="http://customersproject"
 name="CustomerIdentification_Service"
 targetNamespace="http://customersproject"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<types>
    <xsd:schema targetNamespace="http://customersproject">
        <xsd:complexType name="CustDetailType">
            <xsd:all>
                <xsd:element name="customerId" type="xsd:string" />
                <xsd:element name="firstName" type="xsd:string" />
                <xsd:element name="lastName" type="xsd:string" />
                <xsd:element name="street" type="xsd:string" />
                <xsd:element name="city" type="xsd:string" />
                <xsd:element name="phone" type="xsd:string" />
                <xsd:element name="email" type="xsd:string" />
                <xsd:element name="card" type="xsd:string" />
            </xsd:all>
        </xsd:complexType>
    </xsd:schema>
</types>

<message name="CustDetail_Response">
    <part name="CustDetail_Output" type="tns:CustDetailType" />
</message>

<portType name="CustomerIdentification_Service">
    <operation name="CustomerIdentificationResponse">
        <output message="tns:CustDetail_Response" />
    </operation>
</portType>

<binding name="CustomerIdentification_Binding" type="tns:CustomerIdentification_Service">
    <portlet:binding />
    <operation name="CustomerIdentificationResponse">
        <portlet:action type="standard"
            name="sendCustDetailResponse"
            caption="Action.Caption"
            description="Action.Description"/>
```
Transfer the data to the property broker by creating the customerId command hyperlink. Configure the parameters for this action as shown in Figure 6-11 on page 140.

- Parameter name: com.ibm.portal.propertybroker.action, value: sendCustDetailResponse. Note that this value has to be the same as the action name specify in the wsdl file.

- Parameter name: customerId, value: #{varcustomers.customerId}. This parameter holds the value of the selected customer ID.
Example 6-15 shows the action handler code to put data into the request object. With the customerId parameter created we retrieve the CustomerDetail object of the customer selected and put this information into the request object.

**Example 6-15  Action handler code**

```java
public String doLink1Action() {
    // Type Java code that runs when the component is clicked
    CustomerDetail message = new CustomerDetail();

    String customerId = (String)getRequestParam().get("customerId");
```
if (customerId != null) {
    for (int i = 0; i < getCustomerList().getCustomers().size(); i++) {
        CustomerDetail detail =
            (CustomerDetail)getCustomerList().getCustomers().get(i);
        if (customerId.equals(detail.getCustomerId())) {
            message = detail;
            getRequestScope().put("customerInfo", message);
            break;
        }
    }
    return "";
}

6.3.2 Enabling the target portlet

Enabling the target portlet involves the following steps:

- Describe the properties and actions with WSDL file as shown in Example 6-16. The data type must be the same as the type defined in CustomerIdentification.wsdl file.

Example 6-16 CustomerDetail.wsdl file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:portlet="http://www.ibm.com/wps/c2a"
    xmlns:tns="http://customersproject"
    name="CustomerDetails_Service"
    targetNamespace="http://customersproject"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<types>
    <xsd:schema targetNamespace="http://customersproject">
        <xsd:complexType name="CustDetailType">
            <xsd:all>
                <xsd:element name="customerId" type="xsd:string" />
                <xsd:element name="firstName" type="xsd:string" />
                <xsd:element name="lastName" type="xsd:string" />
                <xsd:element name="street" type="xsd:string" />
                <xsd:element name="city" type="xsd:string" />
                <xsd:element name="phone" type="xsd:string" />
                <xsd:element name="email" type="xsd:string" />
```
You need to create a new class where we implement the processAction() method that will process action requests from the property broker. This class extends of com.ibm.faces.portlet.FacesPortlet and implements the processAction() method as shown in Example 6-17 on page 143. This method retrieves the parameter com.ibm.portal.propertybroker.action to check if the source portlet has sent an action. The action name has to be the same as the action name specifies in CustomerDetails.wsdl file.
Example 6-17  Receiving actions in processAction() method of target portlet

```java
public void processAction(ActionRequest request, ActionResponse response) throws PortletException {
    // Retrieve target action name
    String actionName = (String)request.getParameter("com.ibm.portal.propertybroker.action");
    if (actionName != null) {
        // Invoke custom processing only if property broker action
        if (actionName.equals("receiveCustDetailRequest")) {
            CustomerDetail custDetail =
                (CustomerDetail)request.getAttribute("customerInfo");
            request.getPortletSession().setAttribute("customerInfo", custDetail);
        }
    }
    super.processAction(request, response);
}
```

In the CustomerDetails portlet we have to override the provided portlet class with the CustomerDetailsPortlet class as shown in Figure 6-12.

![Portlet Deployment Descriptor](image)

**Figure 6-12  Override portlet class**
Lastly, you need to modify the generated pagecode CustomerDetailsView.java, to check if the customerInfo attribute is stored in session, that means this portlet has received an action of the property broker and the processAction method has stored the information received in the session.

Example 6-18  CustomerDetailsView.java

```java
public CustomerDetail getCustomerDetail() {
    if (customerDetail == null) {
        CustomerDetail customerInfo =
            (CustomerDetail)getSessionScope().get("customerInfo");
        if (customerInfo != null) {
            customerDetail = customerInfo;
        }
    }
    return customerDetail;
}
```

6.4 Working with composite applications

In “Portlet component scenario” on page 116, it illustrates how to develop, assembly, deploy and wire portlet components to WebSphere Portal. This section describes a new technology that has been incorporated into WebSphere Portal 6 to assembly components in a template and then instantiate them to create different applications.

Templates describe the application behavior: interaction between components, access control to components etc. The template is stored as an XML file in a template catalog to be available to the portal community. Templates can be defined as a series of pages, portlets, and points of variability. An application is an instance of a template, these instances run within an application runtime called composite application infrastructure (CAI). When an application is created there are points in the template that can be overridden to give the application a specific context, these are called the points of variability.

Figure 6-13 illustrates the composite application instantiation process from an existing application template.
Templates contain points-of-variability (PoVs). The PoVs represent user-provided input applied to the template infrastructure during instantiation of the composite application. These PoV values are passed down to the components as parameters during composite application instantiation.

### 6.4.1 Creating templates

To create a new template log into Portal Server as administrator and follow these steps:

1. Select Templates from the Portal Launch menu.
2. Select Template Library page and click New button to create a new template. Enter the following values as shown in Figure 6-14:
   a. Template name: User Reservations
   b. Category: Composite Applications
   c. Description: User Reservations Template
   d. Starting point: Portal Blank Template
3. Click OK.

4. From the Application Template Library select the arrow besides the newly created template to launch the context menu and click **Edit Template Layout**.

5. As we selected **Portal Blank Template** as starting point a page called Blank Page has been created. You can work with this page or create a new page by selecting **New Page** icon. Create a new one named User Reservations.
6. Edit the User Reservations page, to place portlets on it, by clicking the pencil icon.

7. In the **Content** tab of Page Layout window, click **Add portlets** and select the portlets that we installed in “Install the application in WebSphere Portal” on page 133.
   a. UserReservationsProject portlet on the top left column.
   b. CustomerReservations portlet on the bottom left column.
   c. ReservationDetails portlet on the right column.

8. In the **Appearance** tab of the page, select **No Skin** Skin for all portlets.

9. In the **Wire** tab of the page, establish wires as you did in “Assemble the application’s wiring” on page 133.

10. Click **Done** to return to the Application Layout page and click **Save** to save the template.

### 6.4.2 Managing roles

You can define roles in an application template to specify the levels of access that application members have when they work with instances of composite applications created from the template.
To manage roles in your template go to the Template Library page select the arrow besides the User Reservations Template and click Manage Roles. You will see two roles created as shown in Figure 6-16.

![Roles](image)

**Figure 6-16  Managing roles in templates**

You can modify these roles or create a new one clicking New. The New Role window appears as shown in Figure 6-17 on page 149. Take a minute to look the new role options:

1. You can choose to base the new role on an existing role.

2. **Application and Page Access Settings** section. All members of an application can view application pages regardless of their assigned roles. In this section you can add the following permission:
   a. Allow members of this role to edit the application and all pages of the application.
   b. Allow members of this role to manage the membership of the application.

3. **Components Access Settings** section. The access to the components that is controlled first by page access and then by levels of access that are defined for each component in this section. Drop down the boxes to select the level of access for each component.
6.4.3 Creating an application library

You need to create an application based on the template created before.

1. Select **Application Library** tab and select the **New** button. Enter the values showed in Figure 6-18 on page 150.
   a. Name: User Reservations Application.
   b. Template: User Reservations
2. Click Ok and it returns to Application Library tab where you can see your newly application. Click on it to edit the application.

3. A page with the three portlets we added to the template appears as shown in Figure 6-19.

4. You can test the application entering a valid customer number like 1234 as we are working with the administrator user. If you have defined roles in the
template select **Assign Application Members** from the page’s context menu to open the Members portlet. This portlet allows you to assign users and groups to each role you defined in the template to provide levels of user access to the application, its pages and the components of the application. Now when you log into Portal with a different user you will see only the components that this user is allowed to.
Developing portlet components

This chapter describes how to develop portlet components using Rational Application Developer and the Lotus Expeditor Toolkit. This chapter will help you understand what is needed in new and existing portlets in order to be deployed as components in composite applications. We provide step-by-step instructions to create, establish wires, and run portlet components in composite applications.

This chapter exercises these tasks:
- Creating client service portlet projects
- Developing portlets as components of a composite application
- Creating the composite application layout
- Wiring portlet components in a composite application
- Running and testing the composite application in the Lotus Expeditor test environment
7.1 Overview

One of the new features now available in Lotus Expeditor (desktop clients) is the support for portlet applications. The main requirement in Lotus Expeditor is that a client service portlet application must be a valid OSGi bundle.

The Expeditor Toolkit provides portlet development tooling components that allow users develop and debug client services portlet applications based on the OSGi framework, as well as converting existing J2EE server based portlet applications to run on the Expeditor platform.

7.2 Sample scenario

In this scenario a very simple sample composite application is created to send a message and enable portlet communications between two portlet components. In Lotus Expeditor composite applications, this is accomplished through the implementation of the wired model.

Portlet components that need to send a message and listen for message events must register their properties in the property broker. Figure 7-1 illustrates the sample scenario where two portlets communicate via the property broker.
The execution flow for this sample scenario has the following sequence:

- Lotus Expeditor invokes the sample composite application with two portlet components (send and receive message portlet components). The user enters a message text in the send message component and submits the request as shown in Figure 7-2.

- The `processAction()` method in the send message portlet component (source portlet) interrogates if this action is to publish a value to the property broker. This output property must be defined in the source portlet WSDL file.

- The property broker obtains the parameter and identifies the target components. This information must be available in the `<wire>` tag of the `plugin.xml` file.

- The `processAction()` method in the receive message portlet (target portlet) will be invoked and will receive the input property. Again, the input property must be defined in the WSDL file of the target portlet component.

- In this scenario, the JSP in the target portlet reads the value and writes the message as shown in Figure 7-2.

This scenario has one string message. It is important to understand that you can also create other kind of messages and interact with other components.

**Note:** Other important detail is that in this sample scenario, if a message in entered in the target portlet, the same `processAction()` method will also handle the submit request from the keyboard. That is, the action is processed in the same `processAction()` method whether it arrives from the keyboard or from the property broker.
7.3 Creating the SendMessage portlet project

The first step is to create a Client Service Portlet project for the SendMessage portlet component. Start Rational Application Developer with the Lotus Expeditor toolkit and follow these steps:

1. Start the New Project wizard by selecting File → New → Other in the menu bar.

2. Under the Client Services section, select Client Service Portlet Project and click Next. See Figure 7-3.

3. In the Portlet Project window enter the following information as shown in Figure 7-4 and click Next.
   a. Project name: com.ibm.itso.demoapp.sendmessage.portlet
   b. Target Runtime: Client Services v6.1
   c. Portlet API: JSR 168 Portlet
d. Portlet name: SendMessagePortlet

e. Portlet type: Basic Portlet

Figure 7-4 Enter the name and target server of your portlet project

4. In the Portlet Settings window enter the following values as shown in Figure 7-5. Click Next.
   a. Package prefix: com.ibm.itso.demoapp.sendmessage
   b. Class prefix: SendMessagePortlet
c. Here you can also change specific titles for locales. For example, under Locale-specific information section, select a locale then click Edit button and replace SendMessagePortlet with SendMessage in all the fields as shown in Figure 7-5.

![New Client Services Portlet Project](image)

**Figure 7-5**  Enter the portlet settings

5. Click Next.

6. In this scenario, the generated portlet does not need to support Edit mode. Moreover, the generated portlet code will be replaced. Therefore, as illustrated in Figure 7-6, uncheck all the options in the Action and Preferences window and click Next.
7. Take default values in Target Definition window and click Finish to generate the project.
8. If you are prompted to open a Web perspective click **Yes**.

**Note:** You may need to select **Window → Reset Perspective** to set the perspective to its current defaults.

9. Replace the `SendMessagePortlet.java` generated code for the code shown in Example 7-1. This portlet invokes a JSP to receive a string message but it does not communicate with other components.

**Example 7-1  **`SendMessagePortlet.java` code**

```java
package com.ibm.itso.demoapp.sendmessage;
import java.io.*;
import javax.portlet.*;
/**
 * A sample portlet based on GenericPortlet
 */
public class SendMessagePortlet extends GenericPortlet {
    public static final String JSP_FOLDER = "/_SendMessagePortlet/jsp/";
        // JSP folder name
    public static final String VIEW_JSP = "SendMessagePortletView";
        // JSP file name to be rendered on the view mode
    public static final String FORM_SUBMIT = "SendMessagePortletFormSubmit";
        // Action name for submit form
    public static final String FORM_TEXT = "wire_text";
        // Parameter name for the text input
    public static final String ACTION_NAME_PARAM = "ACTION_NAME";
    public static final String SEND_MESSAGE_ACTION = "MessageWireAction";

    /**
     * @see javax.portlet.Portlet#init()
     */
    public void init() throws PortletException{
        super.init();
    }

    /**
     * Serve up the <code>view</code> mode.
     *
     * @see javax.portlet.GenericPortlet#doView
     */
    public void doView(RenderRequest request,
                        RenderResponse response)
                        throws PortletException, IOException {
        // Set the MIME type for the render response
        response.setContentType(request.getResponseContentType());
```

160  **Building Composite Applications**
// Invoke the JSP to render
PortletRequestDispatcher rd =
getPortletContext().getRequestDispatcher(getJspFilePath(request,
   VIEW_JSP));

rd.include(request,response);
}
/**
 * Process an action request.
 *
 * @see javax.portlet.Portlet#processAction
 * (javax.portlet.ActionRequest, javax.portlet.ActionResponse)
 */
public void processAction(ActionRequest request,
   ActionResponse response)
   throws PortletException,
   java.io.IOException {
   // send message to property broker
   String wiredText = null;
   String actionName = request.getParameter(ACTION_NAME_PARAM);
   if(SEND_MESSAGE_ACTION.equals(actionName)){
      wiredText = request.getParameter(FORM_TEXT);
      System.out.println
         ("SendMessagePortlet: message received is " + wiredText);
      // request.setAttribute(FORM_TEXT, wiredText);
   }
}
/**
 * Returns JSP file path.
 *
 * @param request Render request
 * @param jspFile JSP file name
 * @return JSP file path
 */
private static String getJspFilePath(RenderRequest request,
   String jspFile) {
   String markup = request.getProperty("wps.markup");
   if( markup == null )
      markup = getMarkup(request.getResponseContentType());
   return JSP_FOLDER + markup + "/" + jspFile + "." +
   getJspExtension(markup);
}
/**
 * Convert MIME type to markup name.
 *
### Building Composite Applications

```java
private static String getMarkup(String contentType) {
    if ("text/vnd.wap.wml".equals(contentType)) {
        return "wml";
    } else {
        return "html";
    }
}
```

```java
private static String getJspExtension(String markupName) {
    return "jsp";
}
```

10. Save the file.

11. Replace the generated `SendMessagePortletView.jsp` file with the code shown in Example 7-2. This is a very simple JSP to support the portlet View mode.

**Example 7-2  SendMessagePortletView.jsp file**

```jsp
<%@page session="false" contentType="text/html" import="javax.portlet.*,com.ibm.itso.demoapp.sendmessage.*" %>
<%@taglib uri="http://java.sun.com/portlet" prefix="portlet" %>
<portlet:defineObjects/>

<%-
    PortletURL actionUrl = response.createActionURL();
    actionUrl.setParameter(SendMessagePortlet.ACTION_NAME_PARAM, SendMessagePortlet.SEND_MESSAGE_ACTION);
-%>

<form method="POST" action="<%= actionUrl.toString() %>">
    <input name="<%=SendMessagePortlet.FORM_TEXT%>" type="text">
    <input name="<%=SendMessagePortlet.FORM_SUBMIT%>" type="submit" value="Submit">
</form>

Enter a message and submit
</DIV>
```
12. Save all your changes.

7.4 Creating the ReceiveMessage portlet project

In this section you will create a second portlet component for the composite application. Start your Rational Application Developer and follow these steps:

1. Start the New Project wizard by selecting File → New → Other in the menu bar.

2. Under Client Services section, select Client Service Portlet Project and click Next.

3. In the Portlet Project window enter the following information as shown in Figure 7-8 and click Next.
   a. Project name: com.ibm.itso.demoapp.receivemessage.portlet
   b. Target Runtime: Client Services v6.1
   c. Portlet API: JSR 168 Portlet
   d. Portlet name: ReceiveMessagePortlet
   e. Portlet type: Basic Portlet
4. In the Portlet Settings window enter the following values as shown in Figure 7-9. Click Next.

   a. Package prefix: com.ibm.itso.demoapp.receivemessage
   b. Class prefix: ReceiveMessagePortlet
   c. Remove the suffix Portlet from the locale specific titles

Figure 7-8  Enter the name and target server of your portlet project
5. This step is optional. If the portlet needs to support locale specific information, click **Add** under Locale-specific information section and as an example enter the values shown in Figure 7-10 for the language Spanish. Click **OK** and then **Next**.
6. Click Next.

7. As illustrated in Figure 7-6, uncheck all the options in the Action and Preferences window and click **Next**.

8. Take default values in Target Definition window and click Finish to generate the project.
Figure 7-12   Selecting target definition

9. Replace the ReceiveMessagePortlet.java generated code for the code shown in Example 7-3. This portlet invokes a JSP to receive a string message from the keyboard and display it, but it does not communicate with other components.

Example 7-3 ReceiveMessagePortlet.java code

```java
package com.ibm.itso.demoapp.receivemessage;

import java.io.IOException;
import javax.portlet.ActionRequest;
import javax.portlet.ActionResponse;
import javax.portlet.GenericPortlet;
import javax.portlet.PortletException;
import javax.portlet.RequestDispatcher;
import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

/**

*/
```


* A sample portlet based on GenericPortlet

```java
public class ReceiveMessagePortlet extends GenericPortlet {

    public static final String JSP_FOLDER = "/_ReceiveMessagePortlet/jsp/";
    // JSP folder name
    public static final String VIEW_JSP = "ReceiveMessagePortletView";
    // JSP file name to be rendered on the view mode
    public static final String FORM_SUBMIT = "MessageReceived";
    // Action name for submit form
    public static final String FORM_TEXT = "wire_text";
    // Parameter name for the text input
    public static final String ACTION_NAME_PARAM = "ACTION_NAME";
    public static final String RECEIVE_MESSAGE_ACTION = "MessageWireAction";

    public String messageWired = new String();
    /**
     * @see javax.portlet.Portlet#init()
     */
    public void init() throws PortletException{
        super.init();
    }
    /**
     * Serve up the <code>view</code> mode.
     *
     * @see javax.portlet.GenericPortlet#doView
     */
    public void doView(RenderRequest request, RenderResponse response)
        throws PortletException, IOException {
        // Set the MIME type for the render response
        response.setContentType(request.getResponseContentType());
        request.setAttribute("MessageWired", messageWired);
        // Invoke the JSP to render
        PortletRequestDispatcher rd =
            getPortletContext().getRequestDispatcher(getJspFilePath(request, VIEW_JSP));
        rd.include(request, response);
    }
    /**
     * Process an action request.
     *
     * @see javax.portlet.Portlet#processAction
     */
    public void processAction(ActionEvent actionEvent)
        throws PortletException, IOException {
        // Get the action name
        String actionName = actionEvent.getActionName();
        // Get the parameter name for the text input
        String paramName = actionEvent.getParameterName();
        // Get the parameter value
        String parameterValue = actionEvent.getParameter(paramName);
        // Add the text to the message wired
        messageWired += parameterValue;
    }
}
```
public void processAction(ActionRequest request,
    ActionResponse response)
    throws PortletException, java.io.IOException {
        String actionName = request.getParameter(ACTION_NAME_PARAM);

        System.out.println("ReceiveMessagePortlet: action received");

        if(RECEIVE_MESSAGE_ACTION.equals(actionName)) {
            messageWired = request.getParameter(FORM_TEXT);
            System.out.println("ReceiveMessagePortlet: message received is " + messageWired);
        }

        /**
         * Returns JSP file path.
         *
         * @param request Render request
         * @param jspFile JSP file name
         * @return JSP file path
         */
        private static String getJspFilePath(RenderRequest request,
                String jspFile) {
            String markup = request.getProperty("wps.markup");
            if( markup == null )
                markup = getMarkup(request.getResponseContentType());
            return JSP_FOLDER + markup + "/" + jspFile + "." + 
                getJspExtension(markup);
        }

        /**
         * Convert MIME type to markup name.
         *
         * @param contentType MIME type
         * @return Markup name
         */
        private static String getMarkup(String contentType) {
            if( "text/vnd.wap.wml".equals(contentType) )
                return "wml";
            else
                return "html";
        }

        /**
         * Returns the file extension for the JSP file
         *
         * @param markupName Markup name
         */
private static String getJspExtension(String markupName) {
    return "jsp";
}

10. Save the file.

11. Replace the generated ReceiveMessagePortletView.jsp file with the code shown in Example 7-4. This is a very simple JSP to support the portlet View mode.

Example 7-4  ReceiveMessagePortletView.jsp file

```html
<%@page session="false" contentType="text/html"
import="javax.portlet.*,com.ibm.itso.demoapp.receivemessage.*" %>
<%@taglib uri="http://java.sun.com/portlet" prefix="portlet" %>
<portlet:defineObjects/>
<% String messageWired =
   (String)renderRequest.getAttribute("MessageWired");

    PortletURL actionUrl = renderResponse.createActionURL();
    actionUrl.setParameter(ReceiveMessagePortlet.ACTION_NAME_PARAM,
        ReceiveMessagePortlet.RECEIVE_MESSAGE_ACTION);
%>
    <DIV style="margin: 6px">
        <form method="POST" action="<%= actionUrl.toString() %>">
            <input name="<%=ReceiveMessagePortlet.FORM_TEXT%>" type="text">
            <input name="<%=ReceiveMessagePortlet.FORM_SUBMIT%>" type="submit" value="Submit">
        </form>
        <% if( messageWired.length()>0 ) {%>
            <b>The message received was '<%=messageWired%>' </b>
        <% } else {%>
            <b>Enter a message and submit</b>
        <% } %>
    </DIV>
```

12. Save all your changes.
7.5 Creating the DemoApp composite application

From the portlet programmer point of view there should not be major differences related to the portlet code. However, in a Lotus Expeditor environment, a composite application needs to be assembled in order to be deployed and executed.

Creating the application bundle
The composite application is assembled using a separate bundle representing the application itself. Execute the following steps to create the composite application bundle:

1. Select New → Project from the menu bar. Select Client Services → Client Services Project and click Next.

2. Enter com.ibm.itso.demoapp.application as the project name and click Next.

3. Take default values in the Client Services Content window and click Finish to create the project. The wizard will create the project and will open the MANIFEST.MF file for this bundle.
Configuring the portlets to run inside the RCP View
Each portlet has to be displayed in the RCP Portlet Viewer that Lotus Expeditor provides. The viewer will render the portlet contents as part of a RCP View. To define the viewers for the portlets follow these steps:

1. In the MANIFEST.MF file, go to the Extensions tab and click Add.
2. Uncheck Show only extensions point from the required plug-ins and select com.ibm.rcp.portletviewer.portlets as shown in Figure 7-14.

![Figure 7-14 Adding the com.ibm.portletviewer.portlets extension](image)

3. Click Finish and select No in the New plug-in dependency dialog.
5. Select New -> portletData.
6. Enter the following details, as shown in Figure 7-15, and save the changes.
   a. entityId: com.ibm.itso.demoapp.sendmessage.portlet
   b. portletname: SendMessagePortlet
   c. contextroot: /com.ibm.itso.demoapp.sendmessage.portlet
   d. portletwindowid: default
   e. title: Send Message
7. Right-click com.ibm.rcp.portletviewer.portlets again and select **New -> portletData**. Enter the following details, as shown in Figure 7-16 on page 174 and save the changes.

   a. **entityId**: com.ibm.itso.demoapp.receivemessage.portlet
   
   b. **portletname**: ReceiveMessagePortlet
   
   c. **contextroot**: /com.ibm.itso.demoapp.receivemessage.portlet
   
   d. **portletwindowid**: default
   
   e. **title**: Receive Message
8. Save your files.

9. Select the plugin.xml tab and verify that the plugin.xml file contains the extension point and portlet definitions as illustrated in Example 7-5.

Example 7-5  Updated plugin.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension point="com.ibm.rcp.portletviewer.portlets">
    <portletData
        contextroot="/com.ibm.itso.demoapp.sendmessage.portlet"
        entityId="com.ibm.itso.demoapp.sendmessage.portlet"
        portletname="SendMessagePortlet"
        portletwindowid="default"
        title="Send Message"/>
    <portletData
        contextroot="/com.ibm.itso.demoapp.receivemessage.portlet"
        entityId="com.ibm.itso.demoapp.receivemessage.portlet"
        portletname="ReceiveMessagePortlet"
        portletwindowid="default"
        title="Receive Message"/>
  </extension>
</plugin>
```
Creating the Eclipse perspective to layout the components
When a composite application is defined programatically, instead of using the Portal Tools and NCI, an Eclipse perspective has to be defined by the developer in order to layout the application components. Follow these steps to create the perspective:

1. Edit the Manifest.mf file.
2. On the Extensions tab, click Add, select org.eclipse.ui.perspectives as shown in Figure 7-17 and click Finish.

3. Right-click org.eclipse.ui.perspectives and select New -> perspective. Enter the following values, as shown in Figure 7-18.
   a. id: com.ibm.itso.demoapp.application.perspective
   b. name: DemoApp Perspective
4. Click **class**. The New Java class dialog will show. Enter **DemoAppPerspective** as the class name, as shown in Figure 7-19 and click Finish.
5. Replace the generated code with the code shown in Example 7-6. Save the changes.

Example 7-6  DemoAppPerspective.java

```java
package com.ibm.itso.demoapp.application;
import org.eclipse.ui.IPageLayout;
import org.eclipse.ui.IPerspectiveFactory;
public class DemoAppPerspective implements IPerspectiveFactory {
  private static final String PORTLET_VIEWER_PREFIX =
      "com.ibm.rcp.portletviewer.portletview:";
  private static final String RECEIVE_MESSAGE_VIEWER_ID =
      PORTLET_VIEWER_PREFIX
      + "com.ibm.itso.demoapp.receivemessage.portlet";
  private static final String SEND_MESSAGE_VIEWER_ID =
      PORTLET_VIEWER_PREFIX
      + "com.ibm.itso.demoapp.sendmessage.portlet";
  public void createInitialLayout(IPageLayout pageLayout) {
    pageLayout.addStandaloneView(SEND_MESSAGE_VIEWER_ID, true,
      IPageLayout.TOP, 0.45f, pageLayout.getEditorArea());
    pageLayout.addStandaloneView(RECEIVE_MESSAGE_VIEWER_ID, true,
      IPageLayout.RIGHT, 0.45f, SEND_MESSAGE_VIEWER_ID);
    pageLayout.setEditorAreaVisible(false);
  }
}
```

6. Go back to the MANIFEST.MF's Extensions tab. Verify that the perspective entry looks as shown in Figure 7-20.

![Extension Element Details](image)

**Figure 7-20 Perspective details**

7. Browse the plugin.xml and review the new entries.
Defining the composite application
You also need to define the application launch options. Here you can define, for example, the initial perspective ID to be opened when the application is started, and the application label and icon to be shown in Lotus Expeditor launcher. Follow these steps:

1. Open the com.ibm.itso.demoapp.application's plugin.xml file and click Add.
2. Uncheck Show only extensions point from the required plug-ins.
4. Enter the following details for the com.ibm.eswe.workbench.wctApplication extension, as shown in Figure 7-21 and save the changes.
   a. ID: com.ibm.itso.demoapp.application
   b. Name: DemoApp Application

   ![Figure 7-21 Extension point definition](image)

5. Right-click com.ibm.eswe.workbench.wctApplication and select New -> application. Enter the following details, as shown in Figure 7-22 and save the changes.

   ![Figure 7-22 DemoApp application definition](image)

6. Verify that the plugin.xml file you created looks as illustrated in Example 7-7.
**Example 7-7  plugin.xml file**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension point="com.ibm.rcp.portletviewer.portlets">
    <portletData
      contextroot="/com.ibm.itso.demoapp.sendmessage.portlet"
      entityId="com.ibm.itso.demoapp.sendmessage.portlet"
      portletname="SendMessagePortlet"
      portletwindowid="default"
      title="Send Message"/>
    <portletData
      contextroot="/com.ibm.itso.demoapp.receivemessage.portlet"
      entityId="com.ibm.itso.demoapp.receivemessage.portlet"
      portletname="ReceiveMessagePortlet"
      portletwindowid="default"
      title="Receive Message"/>
  </extension>

  <extension point="org.eclipse.ui.perspectives">
    <perspective
class="com.ibm.itso.demoapp.application.DemoAppPerspective"
    id="com.ibm.itso.demoapp.application.perspective"
    name="DemoApp Perspective"/>
  </extension>

  <extension id="com.ibm.itso.demoapp.application"
    name="DemoApp Application"
    point="com.ibm.eswe.workbench.WctApplication">
    <Application
      DisplayName="DemoAPP"
      PerspectiveId="com.ibm.itso.demoapp.application.perspective"/>
  </extension>
</plugin>
```

### 7.6 Running the composite application

Follow these steps to run and test the DemoAPP composite application using the Lotus Expeditor Test Environment. Component intercommunication has not been implemented yet.

1. If you are in the Web perspective change to the Plug-in development perspective
2. Select **Run -> Run** from the menu bar.

3. In the Create, manage and run configurations window, select **Client Services** and click **New launch configuration** icon.

![Create, manage, and run configurations](image)

*Figure 7-23  Creating new Configuration launcher*

4. Enter the configuration name as **DemoApp Composite Application**. In the Plug-ins tab ensure that the plug-ins shown in Figure 7-24 are selected.
5. Click **Apply** and then click **Run**. After few seconds the Expeditor Client will be launched. If you are prompted for the KeyStore password, enter your password and click **Login** button. If this is the first invocation, you will be requested to confirm your password and you will need to remember it.

6. Select **Open → DemoAPP**

---

**Figure 7-24** Settings for the new Configuration

**Figure 7-25** Open the composite application
7. You should see a perspective, as shown in Figure 7-26, with the SendMessage and ReceiveMessage portlets.

![Figure 7-26  Running the Composite Application](image)

8. The SendMessage portlet receives a text message and sends it to the console for verification. Enter a text message, submit and verify that the entered message is written to the console.

9. The ReceiveMessage portlet displays a text value (echo) with the value entered in the input text of the form. Enter a text message, submit and verify that the entered message is displayed.

10. Close the application and the Expeditor Client. Although both portlets are displayed within the same perspective, they are not aware of each other. In the next section you will enable the Property Broker to establish communications between the two portlets (wiring).

### 7.7 Wiring portlet components

In this section you will update the composite application portlet components to support wiring and intercommunicate with other components. You will need to perform the following tasks:

- For each communicating portlet, create a WSDL file describing what properties will be exchange.
- Update the portlet descriptor file (portlet.xml) with the WSDL file locations
- Update the DemoApp application bundle plugin.xml file to add the property broker extension point and define wires between components
If needed, update portlets to support wiring.

7.7.1 Creating WSDL files

In this section you will create the WSDL files describing the properties to be exchanged.

SendMessage WSDL

Follow these steps to create the WSDL file for the SendMessage portlet:

1. In the Project Explorer view expand the com.ibm.itso.demoapp.sendmessage.portlet project.
2. Right-click the WebContent folder and select New → Folder.
3. In the New Folder window enter wsdl as Folder name. Click Finish.
4. You can create the WSDL by right-clicking on the newly created wsdl folder, selecting New → Other → WebServices → WSDL and using the WSDL editor to fill the properties. However, for simplicity you will import the WSDL files provided for you as additional materials.
5. Right-click the wsdl folder and select Import → General → File System from the context menu and then click Next.
6. The File system import window appears as shown in Figure 7-27.
7. Browse the directory where the wsdl files are located and select the SendMessage.wsdl file. Make sure the Create selected folders only option is checked and click Finish.
8. Review the SendMessage.wsdl file shown in Example 7-8 and notice the following observations:

   - Defines a data type called WireStringType in the namespace of http://www.ibm.com/wps/c2a that has the restrictions of xsd:string.
   - Defines a message called WireResponse that has a data item, wire_msg_text of type WireStringType.
   - Defines a port type name Wire_Service that has an operation named WireOperation and contains an output message of type WireResponse.
   - Declares a binding called SendMessageBinding of port type Wire_Service containing an operation of type WireOperation.
   - An action is declared named MessageWireAction.
   - For JSR 168 portlets the type is standard. The action name parameter is ACTION_NAME. This action defines the output data item to be exchanged (wire_text).

Example 7-8  WSDL file for SendMessagePortlet

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="OrderDetail_Service"
    targetNamespace="http://www.ibm.com/wps/c2a"
    xmlns="http://schemas.xmlsoap.org/wsd1/"
```
xmlns:soap="http://schemas.xmlsoap.org/wsd1soap/
xmlns:tns="http://www.ibm.com/wps/c2a"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:portlet="http://www.ibm.com/wps/c2a">
  <types>
    <xsd:schema targetNamespace="http://www.ibm.com/wps/c2a">
      <xsd:simpleType name="WireStringType">
        <xsd:restriction base="xsd:string">
          </xsd:restriction>
      </xsd:simpleType>
    </xsd:schema>
  </types>

  <message name="WireResponse">
    <part name="wire_wsdl_text" type="portlet:WireStringType"/>
  </message>

  <portType name="Wire_Service">
    <operation name="WireOperation">
      <output message="tns:WireResponse"/>
    </operation>
  </portType>

  <binding name="SendMessageBinding" type="portlet:Wire_Service">
    <portlet:binding/>
    <operation name="WireOperation">
      <portlet:action name="MessageWireAction" type="standard" actionNameParameter="ACTION_NAME"/>
      <output>
        <portlet:param name="wire_text" partname="wire_wsdl_text" boundTo="request-attribute"/>
      </output>
    </operation>
  </binding>
</definitions>

**ReceiveMessage WSDL**

Follow these steps to create the WSDL file for the ReceiveMessage portlet:

1. In the Project Explorer view expand the
   com.ibm.itso.demoapp.receivemessage.portlet project.
2. Right-click the WebContent folder and select **New → Folder**.
3. In the New Folder window enter `wsdl` as Folder name. Click Finish.

4. Right-click the `wsdl` folder and select **Import → General → File System** from the context menu and then click Next.

5. The File system import window appears as shown in Figure 7-28. Browse the directory where the `wsdl` files are located and select the `ReceiveMessage.wsdl` file. Make sure the **Create selected folders only** option is checked and click **Finish**.

![Figure 7-28 Importing wsdl files](image)

6. Review the `ReceiveMessage.wsdl` file and notice the following observations:

   - Defines a data type called `WireStringType` in the namespace of `http://www.ibm.com/wps/c2a` which has the restrictions of `xsd:string`.
   
   - Defines a message called `WireRequest` that has a data item, `wire_wSDL_text` of type `WireStringType`.
   
   - Defines a port type name `Wire_Service` that has an operation named `WireOperation` and contains an input message of type `WireRequest`.
   
   - Declares a binding called `ReceiveMessageBinding` of port type `Wire_Service` containing an operation of type `WireOperation`.
   
   - An action is declared named `MessageWireAction`.
   
   - For JSR 168 portlets its type is standard.
– Its action name parameter is `ACTION_NAME`. This action declares the input data item named `wire_text`.

Example 7-9  WSDL file for ReceiveMessagePortlet

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="OrderDetail_Service"
    targetNamespace="http://www.ibm.com/wps/c2a"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/
    xmlns:tns="http://www.ibm.com/wps/c2a"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:portlet="http://www.ibm.com/wps/c2a">
    
    <types>
        <xsd:schema targetNamespace="http://www.ibm.com/wps/c2a">
            <xsd:simpleType name="WireStringType">
                <xsd:restriction base="xsd:string">
                    <xsd:restriction/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:schema>
    </types>

    <message name="WireRequest">
        <part name="wire_wsdl_text" type="portlet:WireStringType"/>
    </message>

    <portType name="Wire_Service">
        <operation name="WireOperation">
            <input message="tns:WireRequest"/>
        </operation>
    </portType>

    <binding name="ReceiveMessageBinding" type="portlet:Wire_Service">
        <portlet:binding/>
        <operation name="WireOperation">
            <portlet:action name="MessageWireAction" type="standard"
                actionNameParameter="ACTION_NAME"/>
            <input>
                <portlet:param name="wire_text" partname="wire_wsdl_text"/>
            </input>
        </operation>
    </binding>
</definitions>
```
7.7.2 Updating the portlet.xml file

A portlet component implementing the wsdl file for communications needs to include a reference to its WSDL file. This reference (location) must be defined in the portlet deployment descriptor (portlet.xml) preference parameter called com.ibm.portal.propertybroker.wsdllocation.

**SendMessage portlet**

To update the portlet.xml file for the SendMessage portlet, follow these steps:

1. In the Project Explorer view, expand
   com.ibm.itso.demoapp.sendmessage.portlet/WebContent/WEB-INF and double-click in portlet.xml
2. In the Source tab scroll down to the </portlet-info> tag of the SendMessage portlet and add the location of the WSDL file as a portlet preference as illustrated in Example 7-10.

   **Example 7-10 Add portlet preferences in SendMessagePortlet**

   ```xml
   </portlet-info>
   <portlet-preferences>
     <preference>
       <name>com.ibm.portal.propertybroker.wsdllocation</name>
       <value>/wsdl/SendMessage.wsdl</value>
     </preference>
   </portlet-preferences>
   </portlet>
   ```

**ReceiveMessage portlet**

In a similar way, update the portlet.xml file for the ReceiveMessage portlet, follow these steps:

1. In the Project Explorer view, expand
   com.ibm.itso.demoapp.receivemessage.portlet/WebContent/WEB-INF and double-click in portlet.xml
2. In the Source tab scroll down to </portlet-info> tag of the ReceiveMessage portlet as illustrated in Example 7-11.
Example 7-11  Add portlet preferences in ReceiveMessagePortlet

```xml
<portlet-info>
  <portlet-preferences>
    <preference>
      <name>com.ibm.portal.propertybroker.wsdllocation</name>
      <value>/wsdl/ReceiveMessage.wsdl</value>
    </preference>
  </portlet-preferences>
</portlet>
```

3. Save and close the portlet.xml file.

### 7.7.3 Adding the property broker and defining wires

The DemoApp application bundle plugin.xml file contains information related to the Expeditor application properties. Lotus Expeditor will use this file to control the message flow between communicating portlets.

Follow these steps to register wires using the Property Broker interface:

1. In the Project Explorer view expand `com.ibm.itso.demoapp.application` and right-click the plugin.xml file. Select **Open With → Plug-in Manifest Editor**
2. Select the Extensions tab as shown in Figure 7-29. Click **Add**.

![Figure 7-29   New plugin extension](image-url)
3. In the Extension Points Selection uncheck **Show only extension points from the required plug-ins**.

4. Select the extension point `com.ibm.rcp.propertybroker.PropertyBrokerWire` as shown in Figure 7-30.

5. Click Finish. Select **Yes** when asked to add a new plug-in dependency.

6. In the Extensions tab, under **Extension Details**, select the newly extension added and enter `com.ibm.rcp.portlet.wire` in the ID field and **Portlet Wire** in the Name field.
7. In the All Extensions section, right-click: `com.ibm.rcp.propertybroker.PropertyBrokerWire` and select New → Wire. In the Extension Element Details section, enter the values as shown in Figure 7-32.

8. Save your files.
9. Review the entered properties for this plugin as follows:
   
   a. type is the wire type. The type must be PROPERTY_TO_ACTION for inter-portlet communication or portlet-to-eclipse communication.

   b. sourceentityid is the name of the wire source. If the source is a JSR 168 portlet, the value of this field is the URI of the source portlet window. In this example the value is /com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet/default. The source portlet window information can be found in the plugin.xml file in two different tags:
      
      i. <contextRoot>. This tag has a context to call all portlets on this portal application

      ii. <portlet-name>. It has a name of the portlet that you want to call.

      Note: If the source is an eclipse component, the value of this field is the eclipse view ID. The URI is comprised of the context root, portlet name, and portlet window name

   c. targetentityid is the name of the target. If the target is a JSR 168 portlet (basic portlet), the value of this field is the URI of the target portlet window. The URI is comprised of the context root, portlet name, and portlet window name. The process is the same for sourceentityid but in that case for the target. In this example the value is /com.ibm.itso.demoapp.receivemessage.portlet/ReceiveMessagePortlet/default.

   d. targetname is the name of the target action. The value must come from the WSDL of the target portlet, and it is the name attribute of the <portlet:action> element. In this sample the name is MessageWireAction. If the target is an eclipse component, the value of this field is the eclipse view ID.

   e. sourceparam is the name of the source parameter. The value must come from the WSDL of the source portlet or eclipse component, and it is the name of the <portlet:param> element in the output section.

   f. targetparam is the name of the target parameter. The value must come from the WSDL of the target portlet or eclipse component, and it is the name of the <portlet:param> element in the input section.

10. Review the created wire in the plugin.xml file. The wire definition should look as shown in Example 7-2.

Example 7-12  Property Broker extension point and generated wire

```xml
<extension id="com.ibm.rcp.portlet.wire"
   name="Portlet Wire"
   point="com.ibm.rcp.propertybroker.PropertyBrokerWire">
   <wire
       enable="true"
```
7.7.4 Updating the portlet code

In many cases, portlet components must be updated to support wiring. This section clarifies what needs to be done in the actual portlet code to provide component intercommunication.

**Source portlet**

The source portlet must follow the WSDL definition. In this sample scenario, JSR 168 portlets (standard portlets) store the output property in the request object as specified in the associated WSDL file.

*Note:* Review the WSDL file shown in Example 7-8 on page 184.

The property broker obtains this property during the portlet event handling phase. Therefore, the correct place to store this property, as an attribute in the request object, is in the processAction() method. For example, perform the following steps:

1. From the Project Explorer view select com.ibm.itso.demoapp.sendmessage.portlet/Java Resources.
2. Select src/com.ibm.itso.demoapp.sendmessage and double-click SendMessagePortlet.java to open the file.
3. In the processAction() method, add the statement to save the output property as an attribute in the request object as highlighted in Example 7-13

   **Example 7-13 Updating the processAction() method**

   ```java
   public static final String FORM_TEXT = "wire_text";
   public static final String ACTION_NAME_PARAM = "ACTION_NAME";
   public static final String SEND_MESSAGE_ACTION = "MessageWireAction";
   ..........
   public void processAction(ActionRequest request, ActionResponse response) throws PortletException, java.io.IOException {
      //* send message to property broker
   ```
String wiredText = null;
String actionName = request.getParameter(ACTION_NAME_PARAM);

if(SEND_MESSAGE_ACTION.equals(actionName)){
    wiredText = request.getParameter(FORM_TEXT);
    System.out.println("SendMessagePortlet: message received is "+
                       wiredText);
    request.setAttribute(FORM_TEXT, wiredText);
}

---

**Target portlet**
The target portlet must include support to handle the action as defined in its associated WSDL file. In this sample scenario, the action defined is exactly the same action that processes the submit request and the property broker will pass this action. Therefore, in this case, there is no need to update the processAction() method in the target portlet,

The processAction() method in the target portlet will be invoked and will receive the input property. This property must be defined in the WSDL file of the target portlet component.

---

### 7.8 Running and testing the composite application

Follow these steps to test the Composite Application:

1. Select Run → Run from the menu bar.
2. In the next window select DemoApp Composite Application configuration under Client Services section and click Run.
3. If asked, enter your KeyStore password and click Login.
4. Open the DemoApp application by selecting Open → DemoAPP.
5. You should see a perspective with SendMessage and ReceiveMessage portlets.
6. Enter a value in the input text field of the SendMessage portlet and check that this value is received and displayed by the ReceiveMessage portlet JSP.
7. Close the application and the Expeditor Client.

7.9 Troubleshooting

The following is a check list to help you find a problem:

- Review the WSDL files for errors.
- Make sure that WSDL file locations have been properly defined in the portlet deployment descriptor (portlet.xml).
- Verify that the following extension points are defined in the composite application plugin.xml file:
  a. `<extension point="com.ibm.rcp.portletviewer.portlets"/>
  b. `<extension point="org.eclipse.ui.perspectives">
  c. `<extension point="com.ibm.eswe.workbench.WctApplication">
  d. `<extension point="com.ibm.rcp.propertybroker.PropertyBrokerWire">
- Review the wire definitions in the plugin.xml file. For example, the sample scenario described in this chapter defines the wire tag illustrated in Example 7-14.

Example 7-14 Wire definition

```
<wire enable="true"
  sourceentityid="/com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet/default"
  sourcename="wire_text"
  targetentityid="/com.ibm.itso.demoapp.receivemessage.portlet/ReceiveMessagePortlet/default"
  targetname="MessageWireAction"
```
Make sure that the `processAction()` method in the source portlet provides the output property as an attribute in the request object. The WSDL file should also define this for the property broker as a standard process for JSR 168 portlet components. For example:

**Example 7-15  Publishing the output property during portlet event processing**

```java
public void processAction(ActionRequest request,
                           ActionResponse response)
    throws PortletException,
          java.io.IOException {

    // send message to property broker
    String wiredText = null;
    String actionName = request.getParameter(ACTION_NAME_PARAM);
    if(SEND_MESSAGE_ACTION.equals(actionName)) {
        wiredText = request.getParameter(FORM_TEXT);
        System.out.println("SendMessagePortlet: message received is "
                                  + wiredText);

        request.setAttribute(FORM_TEXT, wiredText);
    }
}
```

Make sure the target portlet provides support in the `processAction()` method to process the defined action. The action name is defined in the portlet WSDL file.
An overview of RCP components

This chapter provides an overview of Rich Client Platform (RCP) applications that use the Standard Widget Toolkit (SWT) and the JFace User Interface framework.

The Standard Widget Toolkit (SWT) is the software component that delivers native widget functionality for the Lotus Expeditor Client platform.

The base SWT widget library offers an initial set of controls, but the JFace framework has been specifically designed to support more complex user interface operations. This chapter provides an overview of the JFace User Interface framework and how it is integrated with Lotus Expeditor.
8.1 SWT overview

SWT is an open source widget toolkit for Java designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented.

- SWT is open source. The initial library that became SWT was originally developed by the Object Technology International (OTI) group at IBM, but later it was taken over by the Eclipse Foundation to be part of the Eclipse platform. SWT can also be used stand-alone.

- SWT is a widget toolkit for Java. A widget is a graphical user interface element responsible for interaction with the user. Buttons, lists, and text boxes are all widgets. There are other widget toolkits for Java (Swing, for example).

- SWT is efficient and portable. SWT widgets directly access the operative system resources using a thin abstraction layer, making SWT applications portable between platforms.

- Because the Lotus Expeditor Client is based on the Eclipse platform, SWT (together with JFace) is available to develop rich client platform applications.

Figure 8-1 outlines the SWT architecture and how it form part of the Lotus Expeditor Client.

What makes SWT unique is that almost every widget in the toolkit is a thin wrapper for a native widget, making SWT highly efficient. Another benefit is that SWT applications keep the look and feel of the native platform where they are
running. In the case where there is not a corresponding native widget, SWT takes the Swing path, implementing the widget from scratch.

Although SWT uses native widgets, they are isolated by a thin layer that makes SWT applications platform independent. This layer uses Java Native Interface (JNI™) to access native code. Because of this, SWT must provide ports for each platform it supports. Each SWT port comes with a native library used to access the operating system widgets. Ports currently exist for several operating systems.

### 8.1.1 SWT and Swing

When the Eclipse platform was being designed it became very clear that the Java Swing user interface framework that comes integrated with Java would not allow market-competitive solutions to be created. The original goal for Eclipse was to create a competitive tooling platform for Windows and Linux®. That goal has been achieved, as it satisfies the decision to build the low-level user interface widgets so that they map into the native operating system.

The Eclipse team also defined a clean interface between the native operating system and the Java API that would be implemented above it. This enables the SWT native widget code to be easily ported to other platforms. This was another major benefit of the Eclipse platform. Not only would it offer the developer user interfaces but it would offer a programming model and framework above the widgets that would ease the creation of rich and powerful applications.

There is often a debate about the merits of SWT versus Swing. The key is to remember why SWT was developed. First and foremost it was to support performant and competitive user interfaces on a native operating system. Swing, on the other hand, had its history in the Netscape Foundation Classes, which were designed to support rich Internet applications delivered across the network. The good news is that Swing user interface widgets can easily be integrated with and work along side SWT widgets. This allows panels, beans, or just controls to be reused.

For a comparison of SWT and Swing, search the Eclipsepedia. For example, see the following entry related to the question “Is SWT better than Swing?” at:

http://wiki.eclipse.org/index.php/FAQ_Is_SWT_better_than_Swing%3F

### 8.1.2 Display

The Display class connects SWT with the underlying window system. It abstracts the windowing system capabilities into one centralized place. The Display class
has many responsibilities like event processing, access to the current active window, multithreading management, and so on.

The Display class reads and dispatches events through its readAndDispatch() method. The thread that performs this task is called the user-interface thread. Many widget methods can only be called inside this thread.

Stand-alone applications usually create and call to the display event processing methods in the application main() method. For Lotus Expeditor applications, the workbench takes care of all these details.

### 8.1.3 Widgets

A widget is a graphical user interface element responsible for the interaction with the user. Widgets in SWT not only refer to controls, they also represent other elements from the underlying window system, like tree items and so on. Because every SWT widget has a native counterpart and the underlying window system usually does not perform automatic life-cycle management, for SWT applications it is extremely important to dispose unused resources, including widgets.

In SWT every widget derives from the abstract widget class. This class provides event handling methods, creation and disposal methods, and so on. Figure 8-2 shows the widget class hierarchy. Visual controls such as buttons and labels are children of control and thereby widget children. Non-visual elements like item descend from widget too. Although not shown in Figure 8-2, tables, trees, and other advanced controls are also children of widget. These complex controls are also containers and therefore they are composite children.
Figure 8-2  Widget class hierarchy

**Label**

Labels are the simplest widgets in SWT. They represent a non-selectable object that acts as a string, image, or separator. Labels are usually used to adorn other SWT widgets like text fields. This widget is shown in Figure 8-3.

A code snippet, displayed in Example 8-1, shows how this widget is used.
Example 8-1  Label widget example

...  
Label label1 = new Label (parentComposite, SWT.NONE);  
label1.setText ("One");  

Label label2 = new Label (parentComposite, SWT.BORDER);  
label1.setImage(anImage);

...

Button

Buttons are commonly found in desktop applications because desktop computers have pointer devices (mice). They allow users to select them and handle selection events. This widget is shown in Figure 8-4.

Figure 8-4  Button widget

A code snippet, displayed in Example 8-2, shows how to use this widget.

Example 8-2  Button widget example

...  
Button upButton = new Button (parentComposite, SWT.RADIO);  
upButton.setText ("Up");  
Button downButton = new Button (parentComposite, SWT.RADIO);  
downButton.setText ("Down");  
...

Text

Text widgets allow users to edit strings. These strings can range from multi-line text to password fields. Text widgets are selectable, which means a user has control over what characters are selected. If no characters are selected, it is indicated by a caret (^). This widget is illustrated in Figure 8-5.
A code snippet, displayed in Example 8-3, shows how this widget is used.

**Example 8-3 Text widget example**

```java
... Text text1 = new Text(parentComposite, SWT.BORDER | SWT.SINGLE);
        text1.setText("Enter some text");

    Text text2 = new Text(parentComposite, SWT.BORDER | SWT.MULTI);
    text2.setText("Enter some text");

    Text text3 = new Text(parentComposite, SWT.BORDER | SWT.SINGLE);
    text3.setEchoChar('*');
    ...
```

**List**

A list is used to represent a set of strings that can be used in a single or multi-select manner. Methods for adding, selecting, and removing items are provided as part the widget API. This widget is shown in Figure 8-6.
A code snippet, displayed in Example 8-4, shows how this widget is used.

**Example 8-4  List widget example**

```java
...  
  List list1 = new List (parentComposite, SWT.BORDER | SWT.SINGLE);  
  list1.setItems (new String[]{...});  
...  
```

**Combo**

A combo control, also called combo box, is a mix of a text widgets and a list control widget. The list is displayed dynamically, usually by clicking a small icon in the control. A combo can be also read only, disallowing enter text not present in the associated list. This widget is shown in Figure 8-7.

![Combo widget](image)

**Figure 8-7  Combo widget**

A code snippet, displayed in Example 8-5, shows how this widget is used.

**Example 8-5  Combo widget example**

```java
...  
  Combo combo1 = new Combo (parentComposite, SWT.READ_ONLY | SWT.DROP_DOWN);  
  combo1.setItems (new String[]{...});  
...  
```

**Menu**

Menus are one of the most common widgets found on desktop user interfaces. They allow for a list of items to be displayed, giving a user the option to choose an item. This widget is shown in Figure 8-8.
Chapter 8. An overview of RCP components

Figure 8-8 Menu widget

A code snippet, displayed in Example 8-6, shows how this widget is used.

Example 8-6 Menu widget example

```java
...  
Menu menuBar = new Menu(shell, SWT.BAR | radioBehavior);  
shell.setMenuBar(menuBar);  
...  
Menu dropDownMenu = new Menu(shell, SWT.DROP_DOWN | radioBehavior);  
item.setMenu(dropDownMenu);  
...  
```

Note: In Lotus Expeditor applications, menus and toolbars are usually created declaratively and not by using the API.

Table

Tables, also known as grids, provide a way to organize data in rows and columns. Each element in the table is called a cell. SWT tables provide headers and cell editing using embedded controls. This widget is shown in Figure 8-9.
A code snippet, displayed in Example 8-7, illustrates how this widget is used.

**Example 8-7 Table widget example**

```java
... Table table1 = new Table (parentComposite, SWT.SINGLE | SWT.BORDER);

for (int i = 0; i < columnTitles.length; i++) {
  TableColumn tableColumn = new TableColumn(table1, SWT.NONE);
  tableColumn.setText(columnTitles[i]);
}

for (int i=0; i<16; i++) {
  TableItem item = new TableItem (table1, SWT.NONE);
  item.setImage (instance.images [i % 3]);
  item.setText (i, "Index" + i);
}
...
```

**Note:** JFace provides a model-based framework to deal with complex controls such as lists, trees, and tables.
**Tree**

Trees present data ordered in a hierarchical manner. Trees provide a hierarchy of elements, known as tree items. Tree items can be collapsed or expanded, making a tree an appropriate control to display a huge amount of data in a reduced space. This widget is shown in Figure 8-10.

![Tree widget](image)

**Figure 8-10  Tree widget**

A code snippet, displayed in Example 8-8, shows how this widget is used.

**Example 8-8  Tree widget example**

```java
Tree tree1 = new Tree(treeGroup, SWT.SINGLE | SWT.BORDER);

for (int i = 0; i < columnTitles.length; i++) {
    TreeColumn treeColumn = new TreeColumn(tree1, SWT.NONE);
    treeColumn.setText(columnTitles[i]);
}

for (int i = 0; i < 4; i++) {
    TreeItem item = new TreeItem(tree1, SWT.NONE);
    setItemText(item, i, "Node_" + (i + 1));
    if (i < 3) {
        TreeItem subitem = new TreeItem(item, SWT.NONE);
        setItemText(subitem, i, "Node " + (i + 1) + ".1");
    }
}
```

...
8.1.4 Events

The SWT event model is based in the *publish-subscribe* model. To receive the events published by a particular widget, the interested observer has to subscribe to them by registering a *listener*. A listener is a class that implements a well-known interface, according to the event type.

SWT supports two kinds of listeners: *typed* and *untyped*. Typed listeners provide a different interface for each event type like selection events, focus events, and so on. Untyped listeners are capable of managing any event regardless its type, generally using a branch structure like an *if* or switch sentence based on the event type. We prefer typed listeners because the use of untyped listeners leads to messy code. Also, typed listeners are more object oriented.

Table 8-1 shows the typed listeners most commonly used in SWT applications.

<table>
<thead>
<tr>
<th><strong>Listener</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>FocusListener</td>
<td>Provides methods that deal with the events that are generated as controls gain and lose focus</td>
</tr>
<tr>
<td>HelpListener</td>
<td>Provides a method that deals with the event that is generated when help is requested for a control, typically when the user presses F1</td>
</tr>
<tr>
<td>KeyListener</td>
<td>Provides methods that deal with the events that are generated as keys are pressed on the system keyboard</td>
</tr>
<tr>
<td>MenuListener</td>
<td>Provides methods that deal with the hiding and showing of menus</td>
</tr>
<tr>
<td>ModifyListener</td>
<td>Provides a method that deals with the events that are generated when text is modified</td>
</tr>
<tr>
<td>MouseListener</td>
<td>Provides methods that deal with the events that are generated as mouse buttons are pressed</td>
</tr>
<tr>
<td>MouseMoveListener</td>
<td>Provides a method that deals with the events that are generated as the mouse pointer moves</td>
</tr>
<tr>
<td>MouseTrackListener</td>
<td>Provides methods that deal with the events that are generated as the mouse pointer passes (or hovers) over controls</td>
</tr>
<tr>
<td>SelectionListener</td>
<td>Provides methods that deal with the events that are generated when selection occurs in a control</td>
</tr>
<tr>
<td>TreeListener</td>
<td>Provides methods that deal with the expanding and collapsing of tree branches</td>
</tr>
</tbody>
</table>
In Example 8-9, a SelectionListener is added to the buttons to manage their selection events. Listeners are generally implemented as anonymous classes.

**Example 8-9 Listener sample code**

```java
Button upButton = new Button(parentComposite, SWT.RADIO);
upButton.setText("Up");
Button downButton = new Button(parentComposite, SWT.RADIO);
downButton.setText("Down");

/* Add the listeners */
SelectionListener selectionListener = new SelectionAdapter()
    {
        public void widgetSelected(SelectionEvent event)
        {
            if (!((Button) event.widget).getSelection()) return;
            setExampleWidgetAlignment();
        }
    }
    upButton.addSelectionListener(selectionListener);
downButton.addSelectionListener(selectionListener);
```

### 8.1.5 Composites and shells

Composites and its subclasses are widgets that can contain other widgets. A special composite subclass called shell represents the main application window. Shell classes are tied to a particular display.

In the Lotus Expeditor application you work in a higher level of abstraction by using the views and editors that the workbench provides. Similarly to the display class, the workbench creates the application shell instances automatically.

Composites can use layouts to arrange its contained widgets. In Example 8-10, you can see how composites are used.

**Example 8-10 Composite sample code**

```java
Composite composite = new Composite(parentComposite, SWT.NONE);
FillLayout fillLayout = new FillLayout();
fillLayout.type = SWT.VERTICAL;
composite.setLayout(fillLayout);
```
8.1.6 Layouts

Layouts are classes that provide a platform-independent way to automatically position and size widgets inside a container (a composite instance, for example).

Although highly recommended, the use of layouts is optional. You can always position the widgets in an absolute way using the setBounds(), setSize(), and setLocation() widget methods.

SWT provides several predefined layouts, which use several strategies to accommodate the widgets. Let us review some of them.

FillLayout
FillLayout is the simplest layout class. It lays out controls in a single row or column, forcing them to be the same size.
Initially, the controls will all be as tall as the tallest control, and as wide as the widest. FillLayout does not wrap, but you can specify margins and spacing. You might use it to lay out buttons in a task bar or toolbar, or to stack check boxes in a group. FillLayout can also be used when a composite has only one child. Figure 8-11 shows how this layout arranges widgets.

![FillLayout diagram]

**Figure 8-11  FillLayout**

Example 8-11 shows how this layout is used.

### Example 8-11  FillLayout code sample

```java
...  FillLayout fillLayout = new FillLayout ();
    fillLayout.type = SWT.VERTICAL;
    composite.setLayout (fillLayout);
    
    Button button0 = new Button (composite, SWT.PUSH);
    button0.setText ("button0");
    button0.setText ("button0");
    ...
```

**RowLayout**

Instances of this class determine the size and position of the children of a composite by placing them either in horizontal rows or vertical columns within the parent composite.

RowLayout aligns all controls in one row if the type is set to horizontal, and one column if it is set to vertical. It has the ability to wrap, and provides configurable margins and spacing. RowLayout has a number of configuration fields. In
addition, the height and width of each control in a RowLayout can be specified by setting a RowData object into the control using setLayoutData(). Figure 8-12 shows how this layout arranges widgets.

Example 8-12 shows how this layout is used.

**Example 8-12  RowLayout code sample**

```java
...  
RowLayout rowLayout = new RowLayout ();  
rowLayout.type = SWT.VERTICAL;  
composite.setLayout (rowLayout);  

Button button0 = new Button (composite, SWT.PUSH);  
button0.setText ("button0");  
RowData data = new RowData (100, SWT.DEFAULT);  
button0.setLayoutData (data);  

Button button1 = new Button (composite, SWT.PUSH);  
button1.setText ("button1");  
data = new RowData (50, 300);  
button1.setLayoutData (data);  
...  
```

**GridLayout**  
Instances of this class lay out the control children of a composite in a grid.
GridLayout has a number of configuration fields, and the controls it lays out can have an associated layout data object, called GridData. The power of GridLayout lies in the ability to configure GridData for each control in the layout.

The numColumns field is the most important field in a GridLayout. Widgets are laid out in columns from left to right, and a new row is created when numColumns + 1 controls are added to the composite. Figure 8-13 shows how this layout orders the widgets.

The GridLayout is so powerful and simple to use that if you learn only one layout, the GridLayout would be your best option.

![GridLayout Diagram]

**Figure 8-13  GridLayout**

Example 8-13 shows how this layout is used.

**Example 8-13  GridLayout code sample**

```java
... 
GridLayout gridLayout = new GridLayout ();
gridLayout.numColumns = 3;
composite.setLayout (gridLayout);

Button button1 = new Button (composite, SWT.PUSH);
button1.setText ("button1");
data = new GridData ();
data.horizontalAlignment = GridData.END;
```
data.verticalAlignment = GridData.FILL;
button1.setLayoutData (data);

... 

**FormLayout**
The FormLayout controls the position and size of the children of a composite by using *FormAttachments* to configure the left, top, right, and bottom edges of each child. Each side of a child control can be attached to a position in the parent composite, or to other controls within the composite by creating instances of FormAttachment and setting them into the top, bottom, left, and right fields of the child's FormData.

Although the FormLayout is the most powerful and customizable layout, it is also the most complex and difficult to use. We recommend using it only when the other simpler layouts are not suitable for your needs. Figure 8-14 shows how this layout orders the widgets.

![FormLayout Diagram](image)

*Figure 8-14  FormLayout*

Example 8-14 shows how this layout is used.

*Example 8-14  GridLayout code sample*

... 

```
FormLayout formLayout = new FormLayout ();
composite.setLayout (formLayout);

Button button0 = new Button (composite, SWT.PUSH);
button0.setText ("button0");
```


8.1.7 Dialogs

In this section we review some of the dialog classes provided in SWT. Dialogs in SWT are supported through the Dialog class. This class is the abstract superclass of the classes that represent the built-in platform dialogs. A dialog typically contains other widgets that are not accessible through any public API. Although the Dialog class belongs to the org.eclipse.swt.widgets package, a dialog is not a widget. Because the SWT dialog classes use the underlying platform dialogs, their appearance can vary widely between platforms.

**MessageBox**

Instances of this class are used to inform or warn the user that something important has occurred.
The MessageBox class provides several options to specify the nature of the message presented to the user. You can select between info, question, error, and warning icons. You can also select which buttons the dialog will show. Also, the open() method provides a way to get the button selected to close the dialog. Figure 8-15 shows how this dialog looks.

![Image](image_url)

*Figure 8-15  MessageBox dialog*

Example 8-15 shows how to create a MessageBox dialog.

*Example 8-15  MessageBox sample code*

```java
...  
  MessageBox messagebox = new MessageBox(parent.getShell(),
                                      SWT.ICON_INFORMATION | SWT.OK);
  messagebox.setText("Information");
  messagebox.setMessage("This is an important information message");
  int result = messagebox.open();
...
```
FileDialog
This dialog allows the user to navigate the file system and select or enter a file name. Note that the dialog does not verify whether the file exists. This dialog is widely used when you need to save or open a file. Figure 8-16 shows how this dialog looks.

![FileDialog](image)

**Figure 8-16** FileDialog

Example 8-16 shows how to create a FileDialog dialog.

**Example 8-16** FileDialog sample code

```java
... FileDialog fileDialog = new FileDialog(parent.getShell()); String fileName = fileDialog.open(); System.out.println(fileName); ...```

ColorDialog
This dialog allows the user to select a color from a predefined set of available colors. Instead of working directly with the Color class, the ColorDialog class
works with a class called RGB that represents a color based in the three primary colors: red, green, and blue.

ColorDialog does not work directly with color instances because they represent underlaying native resources that would need to be disposed, thereby allowing the developer to choose how to create and instantiate these resources. Figure 8-17 shows how this dialog looks.

![ColorDialog](image)

*Figure 8-17  ColorDialog*

Example 8-17 shows how to create a ColorDialog dialog.

*Example 8-17  ColorDialog sample code*

```java
...  
ColorDialog colorDialog = new ColorDialog(parent.getShell());
RGB myColor = colorDialog.open();
System.out.println("R:" + myColor.red + "G:" +
                  myColor.green + "B:" + myColor.blue);
...  
```
FontDialog

This dialog allows the user to select a font from all available fonts in the system. Similarly to ColorDialog, FontDialog works with a FontData class that holds information about the selected font instead of working with the Font class that is associated to a native resource. Figure 8-18 shows how this dialog looks.

![FontDialog](image)

Figure 8-18  FontDialog

Example 8-18 shows how to create a FontDialog dialog.

```java
// Example 8-18  FontDialog sample code
...
FontDialog fontDialog = new FontDialog (parent.getShell());
FontData myFont = fontDialog.open();
System.out.println(myFont.getName());
...
```

8.1.8 Resources

SWT provides classes to work with other graphics resources besides widgets. Colors, fonts, images, and cursors are resources that need to be used in SWT applications. They have associated native resources, so you will need to dispose of these resources when you no longer need them. SWT provides classes to represent these resources.
Color
This class manages the operating system resources that implement SWT's RGB color model. To create a color you can either specify the individual color components as integers in the range of 0 to 255 or provide an instance of an RGB.

Application code must explicitly invoke the Color.dispose() method to release the operating system resources managed by each instance when those instances are no longer required.

Font
This class manages operating system resources that define how text looks when it is displayed. Fonts may be constructed by providing a device and either name, size, or style information or a FontData object, which encapsulates this data.

Application code must explicitly invoke the Font.dispose() method to release the operating system resources managed by each instance when those instances are no longer required.

Image
This represents graphics that have been prepared for display. That is, they are ready to display on widgets with, for example, Button.setImage().

Application code must explicitly invoke the Image.dispose() method to release the operating system resources managed by each instance when those instances are no longer required.

Cursor
This class manages operating system resources that specify the appearance of the on-screen pointer. To create a cursor you specify the device and either a simple cursor style describing one of the standard operating system provided cursors or the image and mask data for the desired appearance.

Application code must explicitly invoke the Cursor.dispose() method to release the operating system resources managed by each instance when those instances are no longer required.

8.1.9 SWT forms
Eclipse Forms is an optional Rich Client plug-in based on SWT and JFace that provides the support for creating portable Web-style user interfaces across all Eclipse user interface categories.
Based on SWT and JFace, Eclipse Forms were not designed to compete with or offer an alternative to SWT or JFace. The plug-in consists of a few carefully chosen custom widgets, layouts, and support classes to achieve the desired effect when used with SWT and JFace.

**Portable Web-style user interfaces**

An Eclipse Form often looks like a Web page. The fact that each aspect of the form is programmatically accessible at all times makes forms powerful and attractive. Achieving the same flexibility in a browser would require extensive DOM support and often proprietary interactions. Eclipse Forms are portable, being written on top of SWT and JFace.

Across all Eclipse UI categories, Eclipse Forms breaks the mode and therefore certain classes of widgets are expected only in specific Eclipse UI categories such as editors, views, wizards, and dialogs. An Eclipse form can appear in any UI category, expanding development possibilities. The UI developers can use the most appropriate concept for the task.

Eclipse Forms make these rich user interfaces possible with the following elements:

- A concept of a form that is suitable for inclusion in content areas such as views and editors
- A toolkit to manage colors, hyperlink groups, and other aspects of a form, and serve as a factory for many SWT controls
- A new layout manager that lays out controls in a manner similar to an HTML table layout algorithm
- A set of custom controls designed to fit in the form, such as hyperlink, image hyperlink, scrollable composite, and section
- A multi-page editor where most or all of the pages are forms, for example, PDE workbench UI framework

You will find that SWT Forms offers a very elegant and consistent UI framework for displaying and collecting data. Many Smart Client applications that are being built for the Lotus Expeditor client are using this framework.

**8.2 JFace user interface framework**

The JFace name goes back a long way in Eclipse history. JFace was the original name of the Java-based user interface framework that was used internally by IBM development to allow developers to contribute new function to products like VisualAge®, Component Broker, and DB2 in a consistent way. It was this...
framework that became part of the Eclipse project that we know today. The reason for mentioning this is that the original concepts that JFace offered developers in the late 1990’s inside IBM are maintained in the latest release of Eclipse.

We have already mentioned that SWT offers a low level widget programming model that is very close to the native operating system. The problem is that in a modern user interface a higher level of component is required that can offer composite widgets and user interaction frameworks like dialogs, wizards, and preferences. This is the goal of JFace in Eclipse today: to support a model-driven API interface to allow widgets and composite widgets to be controlled in a MVC style and offer the developer a more comprehensive set of widgets.

There is a good design reason why SWT and JFace have been separated in this way. It means that SWT can be kept small and lean and close the machine operating system. The JFace framework can sit on top and offers a consistent interface to the composite and higher value user interface actions required by applications, which can all be driven by model data.

The way in which JFace has delivered this to the developer is through what is know as a viewer. A viewer is a model-based analogue of an SWT widget. This model-based approach can result in less code being written and allows the developer to be more productive. It also allows a more centralized way in which to interact with the view models.

JFace does not offer a view for all widgets. Simple widgets like a button can only be accessed through their SWT APIs. The following is a list of the common viewers:

- Table Viewer
- Tree Viewer
- Table Tree Viewer
- Check Box Tree Viewer
- Check Box Table Viewer

JFace also offers the developer a set of other frameworks that assist in the creation of Smart Client applications. These include:

- Wizard management
- Common dialogs and dialog management
- Preferences management

You will find that these extensions to the core SWT widget set will be invaluable in the construction of powerful and easy-to-use Smart Client applications with Lotus Expeditor.
8.2.1 Viewers

JFace viewers are adapters on the SWT widget set. Like widgets, a viewer can be created on any SWT composite. The JFace viewer framework allows you to map your high-level model content onto the more primitive data types that SWT lists, tables, and tree widgets expect. The data for these widgets can be kept in the model and then synchronized with the user interface.

There is a set of defined viewers within JFace. The Table and Tree Viewers are two of them that are very useful in application construction.

Table
Using a Table Viewer allows you to view multiple columns of information for each element in the table by extending the function of the SWT table widget.

Tree
Using a Tree Viewer is a good example of a composite widget within JFace. It uses a SWT tree widget to display the items and then uses a number of other objects to help it.

**Note:** The JFace Tree Viewer is not intended to be subclassed.

The idea is that a Tree Viewer knows about the root element of the tree that it is going to display. You have to tell it what that object is, of course:

```
TreeViewer: void setInput(Object rootElement)
```

To get started, it asks the root element for its children and then displays them. Then, as the user expands one of the children, the Tree Viewer asks that node for its children, and so on. To do this the Tree Viewer does not talk to the domain objects directly; instead it uses another object called a ContentProvider.

**Implementing a TreeContentProvider**
There are six methods to implement, but generally you can get away with just implementing three of them.

This is how the Tree Viewer asks the content provider for the top-level elements directly beneath the root element:

```
ITreeContentProvider: public Object[] getElements(Object element)
```

And then, whenever it needs the children of a particular element, it uses this:

```
ITreeContentProvider: public Object[] getChildren(Object element)
```
In order to figure out whether a node has any children (and then put a plus sign (+) next to it), the Tree Viewer could ask for the children of the node, and then it could ask how many there are. Just in case your code knows of a quicker way to do this, there is another method that you must implement:

```java
class FileTreeContentProvider implements ITreeContentProvider {
    public Object[] getChildren(Object element) {
        Object[] kids = ((File) element).listFiles();
        return kids == null ? new Object[0] : kids;
    }

    public Object[] getElements(Object element) {
        return getChildren(element);
    }

    public boolean hasChildren(Object element) {
        return getChildren(element).length > 0;
    }

    public Object getParent(Object element) {
        return ((File) element).getParent();
    }

    public void dispose() {
    }

    public void inputChanged(Viewer viewer, Object old_input, Object new_input) {
    }
}
```

As you can see, the content provider does not hold a reference to any domain objects. It is the Tree Viewer that holds on to them itself and passes them as arguments to each method in the content provider.

Example 8-19  Example content provider for a Tree Viewer
Implementing the top-level explorer class

Create a Tree Viewer and then set its content provider to the file tree content provider, and then set the input to a folder. In this case, the folder chosen is the top-level folder in the C: drive.

Note that we are required to return the SWT widget from createContents(). As we mentioned above, the JFace Tree Viewer is not the SWT widget, so we cannot return that. We need to get the actual widget from the Tree Viewer. This is done using the getTree() method.

Example 8-20   Example top-level explorer class

```java
import java.io.*;
import org.eclipse.jface.viewers.*;
import org.eclipse.jface.window.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;

public class Explorer extends ApplicationWindow
{
    public Explorer()
    {
        super(null);
    }

    protected Control createContents(Composite parent)
    {
        TreeViewer tv = new TreeViewer(parent);
        tv.setContentProvider(new FileTreeContentProvider());
        tv.setInput(new File("C:\"));
        return tv.getTree();
    }

    public static void main(String[] args)
    {
        Explorer w = new Explorer();
        w.setBlockOnOpen(true);
        w.open();
        Display.getCurrent().dispose();
    }
}
```

The Tree Viewer example creates a view similar to the one shown in Figure 8-19.
8.2.2 Dialogs

Dialogs offer the application the ability to present a separate window that has complete focus and control of the user to assist with supplying or presenting information for the application. Dialogs are used for common tasks as well. The following are the standard dialogs included in JFace.

- DirectoryDialog will allow a file system directory to be selected.
- FileDialog will allow a file to be selected.
- ListDialog will present a selection and then the result returned.
- MessageDialog is used to display information, warning, and error messages.
- InputDialog is used to prompt for a single item of data.
- ProgressMonitorDialog can be used during a long-running process to present information back to the user.
- ErrorDialog can present one or more errors back to the user.
- ColorDialog can be used to select a color from the operating system color palette.

You can see that these common dialogs can be used within Lotus Expeditor applications, offering a consistent look and feel and user experience. They also do not require a lot of coding effort to get them to display.

Example 8-21 illustrates an example of opening the FileDialog.
Example 8-21  Example of opening a file dialog

```java
final Text t = new Text(s, SWT.BORDER | SWT.MULTI);
final Button b = new Button(s, SWT.PUSH | SWT.BORDER);
b.setText("Change Color");
b.addSelectionListener(new SelectionAdapter() {
    public void widgetSelected(SelectionEvent e) {
        ColorDialog cd = new ColorDialog(s);
        cd.setText("ColorDialog Demo");
        cd.setRGB(new RGB(255, 255, 255));
        RGB newColor = cd.open();
        if (newColor == null) {
            return;
        }
        t.setBackground(new Color(d, newColor));
    }
});
```

The color dialog is shown in Figure 8-20.

![Color dialog](image)

Figure 8-20  The color dialog display example

In addition to the standard dialogs, you can create your own. Example 8-22 illustrates a dialog that contains its own customer content area.

Example 8-22  Example customer dialog

```java
KeywordItemDialog dlg = new KeywordItemDialog(getShell());
dlg.setTitle("Edit Keywords");
dlg.setKeyword(strSelKeyword);
dlg.setDescription(strSelDescription);
if(dlg.open()) {
```
Example 8-23 illustrates a sample code to implement the custom dialog.

Example 8-23  A Custom dialog for editing information

```java
package com.ibm.sametime.sample.preferences;

import org.eclipse.swt.SWT;
import org.eclipse.swt.events.
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.widgets.*;

public class KeywordItemDialog extends Dialog {

    String strKeyword;
    String strDescription;
    String strTitle;
    Text txtKeyword;
    Text txtDescription;
    Button btnAdd;
    boolean result;

    public KeywordItemDialog(Shell parent, int style) {
        super(parent, style);
        result = false;
    }

    public KeywordItemDialog(Shell parent) {
        this(parent, 0);
    }

    private boolean isValid() {
        boolean rc = false;
        String str = txtKeyword.getText().trim();
        if(str.length() != 0) {
            str = txtDescription.getText().trim();
            if(str.length() != 0) rc = true;
        }
        return rc;
    }
}
```
if(rc)
    btnAdd.setEnabled(true);
else
    btnAdd.setEnabled(false);
return rc;

public boolean open()
{
    Shell parent = getParent();
    final Shell shell = new Shell(parent, 0x10860);
    if(strTitle != null)
        shell.setText(strTitle);
    Label labAcronym = new Label(shell, 0);
    labAcronym.setText("Keyword");
    txtKeyword = new Text(shell, SWT.NONE);
    if(strKeyword != null)
        txtKeyword.setText(strKeyword);
    txtKeyword.addKeyListener(new KeyAdapter() {
        public void keyPressed(KeyEvent e)
        {
            if(e.keyCode == 27)
                shell.dispose();
        }
    });
    txtKeyword.addTraverseListener(new TraverseListener() {
        public void keyTraversed(TraverseEvent e)
        {
            if(e.detail == 4 && isValid())
            {
                strKeyword = txtKeyword.getText().trim();
                strDescription = txtDescription.getText().trim();
                result = true;
                shell.dispose();
            }
        }
    });
    txtKeyword.addModifyListener(new ModifyListener() {
        public void modifyText(ModifyEvent e)
        {
        }
    });
    return shell.open();
}
isValid();

Label labDescription = new Label(shell, 0);
labDescription.setText("Description");
txtDescription = new Text(shell, SWT.NONE);
if(strDescription != null)
txtDescription.setText(strDescription);
txtDescription.addKeyListener(new KeyAdapter() {
    public void keyPressed(KeyEvent e) {
        if(e.keyCode == 27)
            shell.dispose();
    }
});
txtDescription.addTraverseListener(new TraverseListener() {
    public void keyTraversed(TraverseEvent e) {
        if(e.detail == 4 && isValid()) {
            strKeyword = txtKeyword.getText().trim();
            strDescription = txtDescription.getText().trim();
            result = true;
            shell.dispose();
        }
    }
});
txtDescription.addModifyListener(new ModifyListener() {
    public void modifyText(ModifyEvent e) {
        isValid();
    }
});

Composite compBtn = new Composite(shell, 0);
btnAdd = new Button(compBtn, 0);
btnAdd.setText("OK");
btnAdd.addSelectionListener(new SelectionAdapter() { 

    public void widgetSelected(SelectionEvent e) 
    { 
        strKeyword = txtKeyword.getText().trim(); 
        strDescription = txtDescription.getText().trim(); 
        result = true; 
        shell.dispose(); 
    } 

}); 

Button btnCan = new Button(compBtn, 0); 
btnCan.setText("Cancel"); 
btnCan.addSelectionListener(new SelectionAdapter() { 

    public void widgetSelected(SelectionEvent e) 
    { 
        shell.dispose(); 
    } 

}); 
isValid(); 
FormLayout shellLayout = new FormLayout(); 
shellLayout.marginHeight = 5; 
shellLayout.marginWidth = 5; 
shell.setLayout(shellLayout); 
FormData fd = new FormData(); 
fd.top = new FormAttachment(0, 0); 
fd.left = new FormAttachment(0, 0); 
labAcronym.setLayoutData(fd); 
fd = new FormData(); 
fd.top = new FormAttachment(labAcronym, 5); 
fd.left = new FormAttachment(0, 0); 
fd.right = new FormAttachment(100, 0); 
Point ptParentSize = parent.getSize(); 
fd.width = ptParentSize.x / 2; 
txtKeyword.setLayoutData(fd); 
fd = new FormData(); 
fd.top = new FormAttachment(txtKeyword, 10); 
fd.left = new FormAttachment(0, 0); 
labDescription.setLayoutData(fd); 
fd = new FormData(); 
fd.top = new FormAttachment(labDescription, 5); 
fd.left = new FormAttachment(0, 0); 
fd.right = new FormAttachment(100, 0); 
txtDescription.setLayoutData(fd); 
FillLayout fl = new FillLayout(); 
fl.spacing = 10;
compBtn.setLayout(fl);
fd = new FormData();
fd.top = new FormAttachment(txtDescription, 10);
fd.left = new FormAttachment(0, 0);
compBtn.setLayoutData(fd);
Point pt = parent.getLocation();
shell.setLocation(pt.x + 20, pt.y + 20);
shell.pack();
shell.open();
Display display = parent.getDisplay();
while(!shell.isDisposed())
  if(!display.readAndDispatch())
    display.sleep();
  return result;
}

public String getKeyword()
{
  return strKeyword;
}

public void setKeyword(String strKeyword)
{
  this.strKeyword = strKeyword;
}

public String getDescription()
{
  return strDescription;
}

public void setDescription(String strDescription)
{
  this.strDescription = strDescription;
}

public String getTitle()
{
  return strTitle;
}

public void setTitle(String strTitle)
{
  this.strTitle = strTitle;
}
Figure 8-21 shows the dialog in action.

![Edit Keywords dialog](Image)

**Figure 8-21  Edit Keywords dialog**

### 8.2.3 Wizards

Wizards are used extensively throughout Eclipse. You can use wizards to create a new Java class or new resources like projects, folders, or files. A well-designed wizard can considerably simplify user tasks and increase productivity.

Wizards are meant to take the hassle out of standard, repetitive, or tedious user tasks. For example, the Java New Class wizard can collect enough information to generate a skeleton implementation of a user's class, including package statements, constructors, inherited methods, and other details. Of course, as the wizard developer, you must implement the code that makes the wizard useful for your domain.

Not only does the platform contain many wizards, but there is a lot of support for writing your own. The JFace wizard framework lets you concentrate on the specifics of your wizard implementation. You will need to use the org.eclipse.jface.wizard package of JFace. It is very easy to get started while the support is flexible enough to allow you to add more complex logic to your wizards.
Figure 8-22  Example Eclipse wizard

The following is a simple wizard for collecting a simple form of information and then deciding on the next page to display based on the selection from the first page.

The first step is to launch the wizard from within your application. This can be done within an action or with a selection event on a button.

Example 8-24  Opening a wizard

```java
AddKeywordWizard wizard = new AddKeywordWizard();
WizardDialog dlg = new WizardDialog(cmpParent.getShell(), wizard);
int rc = dlg.open();
if (rc == IDialogConstants.OK_ID)
{
    Keyword key = wizard.getKeyword();
    addEntry(key);
}
```

The AddKeywordWizard now needs to be implemented, and this can be done using the sample code shown in Example 8-25.

Example 8-25  Implementing the AddKeywordWizard

```java
import org.eclipse.jface.dialogs.MessageDialog;
import org.eclipse.jface.wizard.Wizard;
import com.ibm.sametime.sample.extend.Keyword;
import com.ibm.sametime.sample.extend.KeywordType;
```
public class AddKeywordWizard extends Wizard {

    AddKeywordWizardPage main;
    AddKeywordWizardSheetPage sheet;
    AddKeywordWizardSearchPage search;

    public AddKeywordWizard() {
        // Define the pages
        main = new AddKeywordWizardPage();
        search = new AddKeywordWizardSearchPage();
        sheet = new AddKeywordWizardSheetPage();

        // Add them to the UI
        addPage(main);
        addPage(search);
        addPage(sheet);

        setWindowTitle("Add Keyword Wizard");
    }

    Keyword keyword = new Keyword();

    public boolean performFinish() {
        // Build Keyword Object from Wizards
        keyword = main.getKeyword();
        if (keyword.getType().getId().equals(KeywordType.KEYWORD_TYPE_SHEET))
            keyword.setSheet(sheet.getSheet());

        if (keyword.getType().getId().equals(KeywordType.KEYWORD_TYPE_SEARCH))
            keyword.setSearchEngine(search.getSearch());

        // Construct the Keyword Object ready to add to Table
        return true;
    }

    public Keyword getKeyword() {
        return keyword;
    }

    public boolean performCancel() {
        boolean ans = MessageDialog.openConfirm(getShell(), "Add Keyword", 
            "Are you sure to cancel adding a new keyword?");
        if (ans)
            return true;
        else
return false;
}
}

The Wizard controller manages the pages that are displayed and in what order the wizard pages are viewed.

Example 8-26 illustrates sample code of a wizard page.

Example 8-26  Example main wizard page

```java
package com.ibm.sametime.sample.wizard;

import java.util.ArrayList;
import java.util.Iterator;
import org.eclipse.jface.wizard.IWizardPage;
import org.eclipse.jface.wizard.WizardPage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.events.ModifyEvent;
import org.eclipse.swt.events.ModifyListener;
import org.eclipse.swt.layout.FillLayout;
import org.eclipse.swt.widgets.Composite;
import com.ibm.sametime.sample.extend.Keyword;
import com.ibm.sametime.sample.extend.KeywordActivator;
import com.ibm.sametime.sample.extend.KeywordType;
import com.ibm.sametime.sample.wizard.panels.KeywordPanel;

public class AddKeywordWizardPage extends WizardPage implements ModifyListener {

    /**
     * Create the wizard
     */
    public AddKeywordWizardPage() {
        super("Main");
        setTitle("Add Keyword");
        setDescription
            ("Define a keyword that will be used to reference your desktop data");
    }

    KeywordPanel keypanel;

    public void createControl(Composite parent) {
        Composite container = new Composite(parent, SWT.NULL);
        container.setLayout(new FillLayout());
        //
setControl(container);

keypanel = new KeywordPanel(container, SWT.NONE);
setPageComplete(false);

keypanel.getKeyword().addModifyListener(new ModifyListener() {

    public void modifyText(ModifyEvent e) {

        // Clear Error Messages
        setErrorMessage(null);

        // Let's check if the word has spaces
        String ktext = keypanel.getKeyword().getText();
        if (ktext.indexOf(" ") > 0) {
            setErrorMessage("Enter a keyword without spaces");
        } else {

            // Let's check if it exists
            if (KeywordActivator.getDefault().checkKeywords("@" + ktext) != null) {
                setErrorMessage("Keyword @" + ktext + " has already been defined");
            } else {
                // Let's do a final check
                setPageComplete(keypanel.isComplete());
            }

        }
    }

});
keypanel.getType().addModifyListener(this);
keypanel.getDescription().addModifyListener(this);

}

generic Keyword getKeyWord() {

    return keypanel.getKeywordObject();
}

public IWizardPage getNextPage() {

    String nextpage = "Main";
    // From the Type Select the Next Page
    if (!keypanel.getType().getText().equals("")
    {
        KeywordType type =
KeywordActivator.getDefault().getTypeByName(keypanel.getType().getText());

nextpage = type.getName();

// Mark all the other pages complete
ArrayList types = KeywordActivator.getDefault().getTypes();

// Work through wizard pages and say they are all free to go besides
// the one I am working on
Iterator i = types.iterator();
WizardPage next = (WizardPage)getWizard().getPage(nextpage);
WizardPage prev = (WizardPage)getWizard().getPage("Main");
while(i.hasNext())
{
    KeywordType ty = (KeywordType)i.next();
    if(!ty.getName().equals(nextpage))
    {
        WizardPage page =
            (WizardPage)getWizard().getPage(ty.getName());
        // Check its not the current page and the next one
        if (page != null)
        {
            if (page != next && page != prev)
            {
                page.setPageComplete(true);
            }
        }
    }
}

return getWizard().getPage(nextpage);
}

class)
public void modifyText(ModifyEvent text) {
    setPageComplete(keypanel.isComplete());
}

}

The user interface widgets for the wizard page is separated out into a different
class so that it can be used within a standard dialog if the user wants to edit the
information rather than progress through another wizard. This is a design best
practice.

*Example 8-27* User interface composite used to display the content for the wizard page

```java
package com.ibm.sametime.sample.wizard.panels;

import java.util.ArrayList;
```
import java.util.Iterator;
import org.eclipse.swt.SWT;
import org.eclipse.swt.layout/FormAttachment;
import org.eclipse.swt.layoutFormData;
import org.eclipse.swt.layout/FormLayout;
import org.eclipse.swt.widgets.Combo;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Group;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
import com.ibm.sametime.sample.extend.Keyword;
import com.ibm.sametime.sample.extend.KeywordActivator;
import com.ibm.sametime.sample.extend.KeywordType;
import com.swtDesigner.ResourceManager;
import com.swtDesigner.SWTResourceManager;

public class KeywordPanel extends Composite {

    private Text keyword;
    private Text description;
    privateCombo type;
    privateKeyword key = newKeyword();

    /**
     * Create the composite
     * @param parent
     * @param style
     */
    public KeywordPanel(Composite parent, int style) {
        super(parent, style);
        setLayout(new FormLayout());

        finalGroup parametersGroup = newGroup(this, SWT.NONE);
        parametersGroup.setFont(SWTResourceManager.getFont("", 10, SWT.NONE));
        finalFormData formData_1 = newFormData();
        formData_1.right = newFormAttachment(100, -5);
        formData_1.bottom = newFormAttachment(0, 126);
        formData_1.top = newFormAttachment(0, 1);
        formData_1.left = newFormAttachment(0, 1);
        parametersGroup.setLayoutData(formData_1);
        parametersGroup.setLayout(new FormLayout());
        parametersGroup.setText("Keyword Details");

        keyword = new Text(parametersGroup, SWT.BORDER);
    }
}
final FormData formData_3_3 = new FormData();
formData_3_3.bottom = new FormAttachment(0, 33);
formData_3_3.top = new FormAttachment(0, 11);
formData_3_3.right = new FormAttachment(0, 258);
formData_3_3.left = new FormAttachment(0, 103);
keyword.setLayoutData(formData_3_3);
keyword.setToolTipText("Enter a keyword that best represents the service action you will invoke");
keyword.setTextLimit(260);

final Label itemlab = new Label(parametersGroup, SWT.NONE);
final FormData formData_2 = new FormData();
formData_2.top = new FormAttachment(0, 12);
formData_2.left = new FormAttachment(0, 13);
itemlab.setLayoutData(formData_2);
itemlab.setText("Keyword");

type = new Combo(parametersGroup, SWT.NONE);

// Add Types to Drop Down
ArrayList lTypes = KeywordActivator.getDefault().getTypes();
Iterator i = lTypes.iterator();
while(i.hasNext())
{
    KeywordType kt = (KeywordType)i.next();
    type.add(kt.getName());
}

final FormData formData_4 = new FormData();
formData_4.bottom = new FormAttachment(0, 65);
formData_4.top = new FormAttachment(0, 41);
formData_4.left = new FormAttachment(0, 103);
type.setLayoutData(formData_4);
type.setToolTipText("Select the type of Service you would like to invoke with the keyword");

final Label itemlab_1 = new Label(parametersGroup, SWT.NONE);
final FormData formData_2_1 = new FormData();
formData_2_1.bottom = new FormAttachment(0, 61);
formData_2_1.top = new FormAttachment(0, 45);
formData_2_1.right = new FormAttachment(0, 62);
formData_2_1.left = new FormAttachment(0, 13);
itemlab_1.setLayoutData(formData_2_1);
itemlab_1.setText("Type");

final Label itemlab_2 = new Label(parametersGroup, SWT.NONE);
final FormData formData_2_2 = new FormData();
formData_2_2.bottom = new FormAttachment(0, 90);
formData_2_2.top = new FormAttachment(0, 74);
formData_2_2.right = new FormAttachment(0, 83);
formData_2_2.left = new FormAttachment(0, 13);
itemlab_2.setLayoutData(formData_2_2);
itemlab_2.setText("Description");

description = new Text(parametersGroup, SWT.BORDER);
final FormData formData_3_2 = new FormData();
formData_3_2.bottom = new FormAttachment(0, 94);
formData_3_2.top = new FormAttachment(0, 72);
formData_3_2.right = new FormAttachment(100, -10);
formData_3_2.left = new FormAttachment(0, 103);
description.setLayoutData(formData_3_2);
description.setToolTipText("Enter a description that will help you
describe the keyword action ie. Latest Sales Figures for the last quarter");
description.setTextLimit(260);

final Label label = new Label(parametersGroup, SWT.NONE);
label.setImage
    (ResourceManager.getPluginImage(KeywordActivator.getDefault(),
    "images/keywords.png"));
final FormData formData = new FormData();
formData.top = new FormAttachment(0, 12);
formData.left = new FormAttachment(0, 79);
label.setLayoutData(formData);
formData_4.right = new FormAttachment(keyword, 0, SWT.RIGHT);
final FormData formData_3_1 = new FormData();
formData_3_1.bottom = new FormAttachment(0, 94);
formData_3_1.top = new FormAttachment(0, 72);
formData_3_1.right = new FormAttachment(0, 0);
formData_3_1.left = new FormAttachment(0, 103);

}
public Combo getType()
{
    return type;
}

public void setType(Combo type) {
    this.type = type;
}

public Text getKeyword()
{
    return this.keyword;
}

public void setKeyword(Text keyword) {
    this.keyword = keyword;
}

public void setKeywordObject(Keyword sheet) {
}

public Keyword getKeywordObject()
{
    //
    key.setName("@"+keyword.getText());
    key.setDesc(description.getText());
    key.setType
        (KeywordActivator.getDefault().getTypeByName(type.getText()));

    return key;
}

public boolean isComplete()
{
    boolean complete = true;

    if(keyword.getText() == null || keyword.getText().equals"")
        complete = false;

    if(type.getText() == null || type.getText().equals"")
        complete = false;

    if(description.getText() == null || description.getText().equals"")
        complete = false;

    return complete;
}
The result is a wizard controller that displays the wizard panels shown in Figure 8-23.

![Figure 8-23 Example wizard panels](image)

### 8.2.4 Preferences

It is a very common requirement for a Lotus Expeditor application to store user preference or application preference information. This can allow a more customized user experience within the application. Eclipse has an in-built preferences framework that can be used to make the display and editing of preference data very easy.
Common properties for preferences may include server host names, configuration data that may change from client to client, color choices, and user interface style information.

Preferences panels are all displayed in a common preferences view, which can be opened within Lotus Expeditor by clicking File → Preferences. See Figure 8-24.

A bundle can contribute a new preferences panel that can be selected and displayed in the Preferences dialog.

You can add preferences within an existing bundle plug-in or you can create a new bundle. You may also access the preference at different levels or though an OSGi service.

The first class that needs to be created is a preferences constant class that will hold all the preference field names and make sure they are accessed in a consistent way. See Example 8-28.

Example 8-28   Example preference constants class

package com.itso.mqe.integrator.preferences;

/**
 * Constant definitions for plug-in preferences
 */
public class PreferenceConstants {
    public static final String P_MQEQM = "mqeqm";
This example preference page is holding a set of MQe Queue manager names and the host and port of the MQe Gateway. It is important to keep the preference reference names unique.

The second class to define is a PreferenceInitializer. This is used to create the default values. This class could load the preferences from a central store or from a local properties file. It prevents the bundle from having to handle preference initialization during startup. It will also mean that the preferences are only to be initialized once within a specified workspace.

Example 8-29  Example preference initializer

```java
package com.itso.mqe.integrator.preferences;

import org.eclipse.core.runtime.Preferences;
import org.eclipse.core.runtime.preferences.AbstractPreferenceInitializer;
import com.hsbc.es.mqe.integrator.MQeIntegratorPlugin;

/**
 * Class used to initialize default preference values.
 */
public class PreferenceInitializer extends AbstractPreferenceInitializer {

    public void initializeDefaultPreferences() {

        Preferences store = MQeIntegratorPlugin.getDefault()
            .getPluginPreferences();

        store.setDefault(PreferenceConstants.P_MQEQM, "GATEWAY");
        store.setDefault(PreferenceConstants.P_HOST, "services.itso.ibm.com");
        store.setDefault(PreferenceConstants.P_PORT, "1881");

        store.setDefault(PreferenceConstants.P_MQQM, "ITSO_QM");
    }
}
```
store.setDefault(PreferenceConstants.P_MQQ, "IQ.ACCOUNTS");

In this example the preference values are hard coded, but you can see how the preference store is retrieved from the bundle activator, and then the constants are used to store the values.

8.2.5 Field editors

To make preference page construction easier, the Eclipse platform contains a set of field editors that can be easily added to a preference page. The field editors range from color selectors to file selectors and check boxes.

The final class is the user interface that will display the preferences. In the example we use here, Eclipse offers a JFace class that supports rich input field types like colors, directories, and Boolean values. This preference page also uses very simple form style so you can avoid having to create complete SWT panels to display the preferences.

Example 8-30 GatewayPreference.java page

```java
package com.itso.mqe.integrator.preferences;

import org.eclipse.jface.preference.BooleanFieldEditor;
import org.eclipse.jface.preference.FieldEditorPreferencePage;
import org.eclipse.jface.preference.StringFieldEditor;
import org.eclipse.ui.IWorkbench;
import org.eclipse.ui.IWorkbenchPreferencePage;
import com.hsbc.es.mqe.integrator.MQeIntegratorPlugin;
import com.swtdesigner.ResourceManager;

public class GatewayPreferences
    extends FieldEditorPreferencePage
    implements IWorkbenchPreferencePage {

    public GatewayPreferences() {
        super(FieldEditorPreferencePage.GRID);
        setPreferenceStore
            (MQeIntegratorPlugin.getDefault().getPreferenceStore());
        setDescription("Messaging Integration Preferences");
        setTitle("Messaging Integration Preferences");
        setImageDescriptor
            (ResourceManager.getImageDescriptor(GatewayPreferences.class,
                "queueRemoteSmall.gif"));
```
public void createFieldEditors() {
    addField(new BooleanFieldEditor(PreferenceConstants.P_BOOLEAN,
        "&Trace Messages", getFieldEditorParent()));

    final StringFieldEditor stringFieldEditor = new StringFieldEditor(PreferenceConstants.P_MQEQM, "Gateway Queue Manager",
        getFieldEditorParent());
    stringFieldEditor.setErrorMessage("Enter Gateway Queue Manager");
    stringFieldEditor.setEmptyStringAllowed(false);
    addField(stringFieldEditor);
}

    addField(new StringFieldEditor(PreferenceConstants.P_HOST, "Gateway Host Name",
        getFieldEditorParent()));
}

    addField(new StringFieldEditor(PreferenceConstants.P_PORT, "Gateway Port",
        getFieldEditorParent()));
}

    addField(new StringFieldEditor(PreferenceConstants.P_MQQM, "Destination Queue Manager", getFieldEditorParent()));
}

    addField(new StringFieldEditor(PreferenceConstants.P_MQQ, "Destination Queue", getFieldEditorParent()));
}

public void init(IWorkbench workbench) {
}
Each preference value is added sequentially to the page and is bound to the preference constant value. When the Apply button is clicked, the preferences are stored in the preference store in the application workspace. They can then be accessed through the bundleactivator for the application.

The preference page is extended from the JFace FieldEditorPreferencePage. This class allows the use of field editors to be added to the preferences panel, which simplifies the construction of preferences pages.

Figure 8-25 shows the sample preference page we have been constructing. With the preference page defined and initialized the final step is to add the necessary extension points within the bundle plugin.xml. This will register the preference initializer and preference page with the Lotus Expeditor environment.

Example 8-31 shows the plugin extension points to enable preferences to be added to the preference dialog.

**Example 8-31  Plug-in extension points**

```xml
<extension point="org.eclipse.ui.preferencePages">
  <page class="com.itso.mqe.integrator.preferences.GatewayPreferences" name="Messaging" id="com.itso.mqe.integrator.preferences.GatewayPreferences"/>
</extension>
<extension point="org.eclipse.core.runtime.preferences">
</extension>
```
Preferences allow a the Lotus Expeditor application to be extended and allow a level of customization put into the hands of the user.
Developing RCP components

This chapter describes how to develop RCP components using Rational Application Developer and the Lotus Expeditor Toolkit. This chapter will help you understand what is needed in new and existing RCP applications, such as SWT and SWT/JFace, in order to be deployed as components in composite applications. We provide step-by-step instructions to create, establish wires, and run RCP components in composite applications. These tasks are included in this chapter:

- Developing RCP components
- Adding an RCP component to a composite application
- Enabling the RCP component for intercommunication
- Implementing component communications (wiring)
- Running and testing the composite application
9.1 Sample scenario

In this scenario you will develop an RCP component. The new component will then be added to the DemoApp composite application, explained in Chapter 7, “Developing portlet components” on page 153. The sample scenario is illustrated in Figure 9-1.

![Composite application with a SWT component](image)

**Figure 9-1  Composite application with a SWT component**

The sample scenario has the following characteristics:

- The original DemoApp application is composed of two portlets that interchange messages between using the property broker via wires. For details about this sample composite application see Chapter 7, “Developing portlet components” on page 153.
- The source portlet sends messages through the wire.
- The property broker performs the property matching and sends the message to the target portlet.
- The name for the source portlet is SendMessage and for the target portlet is ReceiveMessage as specified in the portlet descriptor (portlet.xml) file.
The WSDL location, needed for communication using the property broker, is defined as a portlet preference for each portlet.

**Note:** WSDL files are used by Expeditor to indicate what properties are going to be exchanged between components.

The namespace (http://www.ibm.com/wps/c2a) and the property type (WireStringType) are the same in both WSDL files, this is a requisite for proper property sharing.

**Note:** The namespace and the property type should be the same in the WSDL files for components participating in property broker communication.

The sample SWT component will be packaged in a separate bundle called com.ibm.itso.demoapp.swtmessage. The SWT component sends messages to the target portlet (ReceiveMessage portlet) and receives messages from the source portlet (SendMessage portlet).

The sample SWT component is composed of a RCP View called MainView that contains a SWT Composite. The MainView view creates the MainComposite composite.

### 9.2 Importing existing components

In this scenario two existing components are included in the composite application. There are the two portlet components in the DemoApp composite application created in Chapter 7, “Developing portlet components” on page 153.

If you do not have the original DemoApp composite application in your workspace, follow these steps to import it to your workspace:

1. Download the original DemoApp composite application to your hard drive.
2. Select **File -> Import** from the menu bar. Select **General -> Existing Projects into Workspace** and click Next.
3. Choose **Select archive file** and browse to the location of the DemoApp.zip file as shown in Figure 9-2.
4. Select the composite application project as well as the SendMessage and ReceiveMessage components as shown in Figure 9-2. Click Finish to import the projects.
9.3 Creating the SWT component

The sample SWT component will be packaged in a separate bundle called com.ibm.itso.demoapp.swtmessage. The SWT component sends messages to the target portlet (ReceiveMessage portlet component) and receives messages from the source portlet (SendMessage portlet component).

The sample SWT component is composed of a RCP View called MainView that contains a SWT Composite called MainComposite. The MainView view creates the MainComposite composite.

Follow these instructions to create the sample SWT component.

**Create the SWTMessage project**

Follow these steps to create the sample SWT project:

1. Select New -> Project from the menu bar.
2. Select Client Services -> Client Services Project and click Next.
3. Enter `com.ibm.itso.demoapp.swtmessage` as the project name and click Next.

4. Take default values in the Client Services Content window, review the project settings and click Finish to create the project.

5. The wizard will create the project and will open the MANIFEST.MF file for the bundle.

**MainComposite class**

Follow these steps to create the MainComposite class:

1. Select **New -> Other** from the menu bar.

2. Select **Java -> Visual Class** and click **Next**.

3. In the Style section, select **SWT -> Composite**, as shown in Figure 9-3.

4. Enter `com.ibm.itso.demoapp.swtmessage` as the package name.

5. Enter **MainComposite** as the class name.

6. Select the option to generate comments.

7. Click **Finish** to create the MainComposite class. Notice that the wizard also generates the activator for you.
8. Optionally, you may want to use the visual editor to design the class. However, for simplicity in this sample scenario, replace the generated class with the code shown in Example 9-1 and save the changes.

Example 9-1  MainComposite.java

```java
package com.ibm.itso.demoapp.swtmessage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Button;
import org.eclipse.swt.graphics.Point;
import org.eclipse.swt.widgets.Composite;
```
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.Label;
import org.eclipse.swt.widgets.Text;
public class MainComposite extends Composite {
    private Label label = null;
    private Text messageText = null;
    private Button sendButton = null;
    private Label messageLabel = null;
    public MainComposite(Composite parent, int style) {
        super(parent, style);
        initialize();
    }
    private void initialize() {
        this.setSize(new Point(406, 91));
        label = new Label(this, SWT.NONE);
        label.setText("Message:");
        label.setBounds(new Rectangle(15, 15, 46, 16));
        messageText = new Text(this, SWT.BORDER);
        messageText.setBounds(new Rectangle(75, 15, 226, 19));
        sendButton = new Button(this, SWT.NONE);
        sendButton.setText("Send");
        sendButton.setBounds(new Rectangle(315, 15, 76, 23));
        sendButton.addSelectionListener
            (new org.eclipse.swt.events.SelectionListener() {
                public void widgetSelected(
                    org.eclipse.swt.events.SelectionEvent e) {
                    sendMessage();
                }
            });
        messageLabel = new Label(this, SWT.NONE);
        messageLabel.setBounds(new Rectangle(15, 60, 376, 16));
        messageLabel.setText("");
    }
    protected void sendMessage() {
    }
} // @jve:decl-index=0:visual-constraint="10,10"

9. The visual editor should synchronize the changes back to designer. The MainComposite class derives from org.eclipse.swt.Composite. It has widgets
to send and receive messages, as shown in Figure 9-4. The messageLabel widget shows the incoming messages from the SendMessage portlet. The sendButton widget is used to send the text written in the messageText widget to the ReceiveMessage portlet.

Figure 9-4  MainComposite layout

10. Save the files.

**MainView class**

Follow these steps to create the MainView class:

1. Select New -> Other from the menu bar.
2. Select Java -> Visual Class and click Next.
3. Enter MainView as the class name.
4. In the Style section, select RCP -> View, as shown in Figure 9-5.
5. **Click Finish** to create the MainView class.
6. Optionally, you may want to use the visual editor to design the class. However, for simplicity in this sample scenario, replace the generated class with the code shown in Example 9-2 and save the changes. The visual editor should synchronize the changes back to the designer area.

Example 9-2  MainView.java

```java
package com.ibm.itso.demoapp.swtmessage;
import org.eclipse.swt.SWT;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.ViewPart;
public class MainView extends ViewPart {
    public static final String ID =
        "com.ibm.itso.demoapp.swtmessage.MainView";
    private MainComposite top = null;
    public void createPartControl(Composite parent) {
        top = new MainComposite(parent, SWT.NONE);
    }
}
```
public void setFocus() {
}

Extension point definition
Follow these steps to define the extension point in the plugin.xml file:

1. Open the MANIFEST.MF file if not already opened.

2. Select the Extensions tab and click Add. Select org.eclipse.ui.views and click Finish

3. The org.eclipse.ui.views is added to the Extension list. Right-click org.eclipse.ui.views and select New -> view. Enter the following details for the view as shown in Figure 9-6. Save the changes.
   a. id: com.ibm.itso.demoapp.swtmessage.MainView
   b. name: Main View
   c. class: com.ibm.itso.demoapp.swtmessage.MainView
   d. allowMultiple: true (required for wiring)

4. Select the plugin.xml tab and verify that the defined extension point looks as shown in Example 9-3.

Example 9-3 Extension point definition in plugin.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
```
9.4 The composite application

The composite application is assembled using a separate bundle representing the application itself. This bundle contains the perspective classes, definitions for the application and definitions for wires.

9.4.1 Creating the composite application project

In this sample scenario the composite application project has been already created in the original DemoApp composite application.

**Note:** For details on how to create the composite application project see Chapter 7, “Developing portlet components” on page 153.

9.4.2 Laying out the components

The composite application components are layout using an Eclipse perspective. In this sample scenario, the Eclipse perspective class has been already created in the original DemoApp composite application. However, it still needs to be updated with the new RCP component you are adding to the composite application.

**Note:** For details on how to create the Eclipse perspective see Chapter 7, “Developing portlet components” on page 153.

Follow these steps to update the composite application perspective:

1. In the DemoApp composite application project, add the highlighted statements in Example 9-4 to include the RCP component.
2. Save the updates.
Example 9-4  DemoAppPerspective.java

```java
package com.ibm.itso.demoapp.application;
import org.eclipse.ui.IPageLayout;
import org.eclipse.ui.IPerspectiveFactory;
public class DemoAppPerspective implements IPerspectiveFactory {
    private static final String PORTLETVIEWER_PREFIX =
            "com.ibm.rcp.portletviewer.portletview:";
    private static final String SWTMESSAGE_VIEW_ID =
            "com.ibm.itso.demoapp.swtmessage.MainView:1";
    private static final String SEND_MESSAGE_VIEWER_ID = PORTLETVIEWER_PREFIX
            + "com.ibm.itso.demoapp.sendmessage.portlet";
    private static final String RECEIVE_MESSAGE_VIEWER_ID = PORTLETVIEWER_PREFIX
            + "com.ibm.itso.demoapp.receivemessage.portlet";

    public void createInitialLayout(IPageLayout pageLayout) {
        pageLayout.addStandaloneView(SEND_MESSAGE_VIEWER_ID, true,
                IPageLayout.TOP, 0.45f, pageLayout.getEditorArea());
        pageLayout.addStandaloneView(RECEIVE_MESSAGE_VIEWER_ID, true,
                IPageLayout.RIGHT, 0.45f, SEND_MESSAGE_VIEWER_ID);
        pageLayout.addStandaloneView(SWTMESSAGE_VIEW_ID, true,
                IPageLayout.BOTTOM, 0.45f, RECEIVE_MESSAGE_VIEWER_ID);
        pageLayout.setEditorAreaVisible(false);
    }
}
```

9.4.3 Defining the composite application launch options

The composite application launch options are defined in the manifest.mf file. Here you specify the initial perspective ID to be opened when the application starts, as well as the application label and icon to be shown in the Lotus Expeditor launcher. In this sample scenario, the launch options have been already defined in the original DemoApp composite application.

**Note:** For details on how to define the composite application launch options see Chapter 7, “Developing portlet components” on page 153.

9.5 Running and testing the composite application

Follow these steps to run the sample in Lotus Expeditor Test Environment:

1. From the menu bar, select Run -> Run.
2. Select your Client Service configuration or create one. For details on how to create a new configuration see Chapter 7, “Developing portlet components” on page 153.

3. Make sure the application plug-ins are included as shown in Figure 9-7.

![Figure 9-7 Plug-ins included in the Client Services configuration](image)

4. Click **Apply** and then **Run** to launch the Lotus Expeditor test environment.

5. If asked, enter your keystore password or enter and confirm a new password if this is the first time you run the Lotus Expeditor client.

6. Select the **Open -> DemoAPP** from the Lotus Expeditor menu to start the application.

7. Figure 9-8 on page 264 shows the composite application running. The RCP component does not have intercommunication capabilities. Close the Lotus Expeditor Test Environment.
In this section you will update the RCP component and the composite application to support wiring for intercommunication with other components.

9.6.1 Adding component intercommunication support

In this section, you will add intercommunication capabilities to the SWT component.

Follow these steps to add capabilities to send messages to the property broker in the RCP component:

1. Right-click the com.ibm.itso.dempapp.swtmessage project and select Properties.

2. Select Client Services and select Property Broker and Property Broker SWT Feature as shown in Figure 9-9.
3. Click **OK** to save the changes.

**Sending messages to the property broker**

In this section you will update the RCP component to send messages to the property broker. Figure 9-10 illustrates this process. The output property is sent directly to the property broker.

**Note:** The WSDL file is not used in this case but it should be included in the component since it will be required by the tools when assembling the composite application.

Figure 9-10 shows the message flow when an RCP component send a message to the property broker:

1. The RCP component sends a message to property broker
2. The output property is received by property broker
3. The property broker looks at defined wires
4. If wires are found for this property, the property broker notifies all target components

Follow these steps to add capabilities to send messages to the property broker from an RCP component:

1. Right-click the `com.ibm.itso.demoapp.swtmessage` package and select **New -> Interface**.

2. Enter **Constants** as the interface name


![Diagram showing message flow](image-url)
4. Click Finish to create it.

5. Copy the code shown in Example 9-5 and save the changes. This code will include constant definitions for the property name, name space and type.

Example 9-5  Constants.java

```java
package com.ibm.itso.demoapp.swtmessage;

public interface Constants {

    String PROPERTY_NAMESPACE = "http://www.ibm.com/wps/c2a";
    String PROPERTY_NAME = "send_text";
    String PROPERTY_TYPE = "WireStringType";

}
```

6. Save the file.

7. Open the MainComposite class and replace the sendMessage() method with the code shown in Example 9-6. This code sends a message to the property broker to publish the output property.
Example 9-6  MainComposite.send\!M\!essage method

...  
protected void send\!M\!essage() {

    if (messageText.getText().equals("")) {
        MessageDialog.openError(this.getShell(),
            "Message cannot be empty","Enter a message to send");
        return;
    }

    final String message = messageText.getText();

    PropertyController prop = PropertyFactory.createProperty();
    prop.setNamespace(Constants.PROPERTY_NAMESPACE);
    prop.setName(Constants.PROPERTY_NAME);
    prop.setType(Constants.PROPERTY_TYPE);
    PropertyValue propValue =
        PropertyFactory.createPropertyValue(prop,message);
    PropertyValue[] values = new PropertyValue[] { propValue };  

    IV\!I\!ewPart view = SWTHelper.locateView(MainView.ID);
    try {
        SWTHelper.changedProperties(values, view);
    } catch (PropertyBrokerException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }

    messageLabel.setText("The message sent was '" + message + "'");
    messageText.setText("");  
}
...

8. Reorganize the imports, for example by pressing CTRL+SHIFT+O or by selecting Source -> Organize Imports from the menu bar.

9. Save the changes.

10. When an user presses the Send button the button widgetSelected() method invokes the sendMessage() method. Examine the added code and observe that the sendMessage() method performs the following tasks:

    a. The sendMessage() method performs some basic validation in the messageText widget to prevent sending empty messages

    b. Gets the message text from the messageText widget.
c. Uses the PropertyController and PropertyValue classes to define the properties to be exchanged. The constants to create the properties are:

- \textit{PROPERTY_NAMESPACE}: http://www.ibm.com/wps/c2a
- \textit{PROPERTY_NAME}: send_text
- \textit{PROPERTY_TYPE}: WireStringType

d. Notice that \textit{PROPERTY_NAMESPACE} and \textit{PROPERTY_TYPE} must be the same than those defined in the WSDL file of the target component. In this scenario, the ReceiveMessage portlet.

e. The method uses the \textit{SWTHelper.changedProperties()} method to publish the properties to the Property Broker. This method takes as parameters the view and the values to publish. The view is used by the property broker to determine what components will receive the message.

f. The link between the view and the target portlet will defined later in the wire shown in Figure 9-16 on page 278.

\textbf{Receiving messages from the property broker}

In this section, you will do the following:

- Add a new method to receive messages from the property broker. In this sample scenario the \textit{updateMessage()} method will be added.

- Implement an action handler in the RCP component to receive notifications from the property broker.

Figure 9-12 illustrates how messages are received from the property broker.

\textbf{Note}: The WSDL file is read by the property broker to understand how the input property should be passed to the RCP component. In this case, an action handler is implemented to receive the message notification.
Follow these steps to add capabilities to receive messages from the property broker in the RCP component:

1. Open the `MainComposite` class and add the `updateMessage()` method shown in Example 9-7. Save the changes.

   **Example 9-7  MainComposite.updateMessage method**
   
   ```java
   ... 
   public void updateMessage(String message) {
       messageLabel.setText(
           "The message received was '" + message + "'";
   }
   ... 
   ```

2. Open the `MainView` class and add the `updateMessage()` method shown in Example 9-8. This method invokes the `MainComposite.updateMessage()` method shown in Example 9-7. Save the changes.

   **Example 9-8  MainView.updateMessage method**
   
   ```java
   ... 
   public void updateMessage(String message) {
       top.updateMessage(message);
   }
   ... 
   ```

**Adding an action handler to RCP component**

In this section you will add an action handler to the RCP component to receive messages from the property broker. Follow these steps:

1. Right-click the `com.ibm.itso.demoapp.swtmessage` package and select New -> Class.

2. Enter ActionHandler as the class name and `org.eclipse.jface.action.Action` as the superclass, as shown in Figure 9-13.

3. Optionally select Generate comments and click Finish to create the class.
4. Copy the code shown in Example 9-9 to the ActionHandler class and save the changes.

**Example 9-9  ActionHandler.java**

```java
package com.ibm.itso.demoapp.swtmessage;
import org.eclipse.jface.action.Action;
import org.eclipse.swt.widgets.Display;
import org.eclipse.swt.widgets.Event;
import com.ibm.rcp.propertybroker.event.PropertyChangeEvent;
import com.ibm.rcp.propertybroker.swt.SWTHelper;
public class ActionHandler extends Action {
    public void runWithEvent(Event event) {
        if (event instanceof PropertyChangeEvent) {
            final PropertyChangeEvent finalEvent =
            (PropertyChangeEvent) event;
            Display.getDefault().asyncExec(new Runnable() {
                public void run() {
                    String message =
                    (String) finalEvent.getPropertyValue().getValue();
                }
            });
        }
    }
}
```
String targetViewId =
    finalEvent.getWireDefinition().getTargetEntityId();
MainView mainView =
    (MainView) SWTHelper.locateView(targetViewId);
if (mainView != null) {
    mainView.updateMessage(message);
}
}

5. Examine the added code. The runWithEvent() method performs the following
tasks:
   a. Casts the event object to PropertyChangedEvent. This class provides
      information about the message received.
   b. Obtains the message from the event object.
   c. Obtains the target view ID from the event object.
   d. Obtains a reference to the target view using the SWTHelper.locateView()
      method.
   e. Calls the MainView.updateMessage() method which in turn calls the
      MainComposite.updateMessage() method to update the messageLabel
      widget.

   Note: The link between the SendMessage portlet and the SWTMessage
   component is defined as a wire shown in Figure 9-17 on page 279.

Create the WSDL file
The component has to define a WSDL file to declare what input properties the
component will receive. In addition, you will also defined the output properties.

Note: Although output properties do not need to be defined in the WSDL file, it is
a best practice to include them. Managed applications tools, such as Portal Tools
with the Network Client Installer (NCI), requires that output properties should
also be defined in the WSDL file.

In this scenario, the WSDL file is saved in the /wsdl/SWTMessage.wsdl file. Follow
these steps to create the WSDL file:

1. Create a folder named wsdl inside the com.ibm.itso.demoapp.swtmessage
   project. For example, right-click this project, select New -> Folder, enter wsdl
   as the folder name and click Finish to create the folder.
2. Create a file named SWTMessage.wsdl inside the wsdl folder and copy the
content shown in Example 9-10.

Example 9-10  SWTMessage.wsdl file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="OrderDetail_Service"
    targetNamespace="http://www.ibm.com/wps/c2a"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
    xmlns:tns="http://www.ibm.com/wps/c2a"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:portlet="http://www.ibm.com/wps/c2a">

    <types>
        <xsd:schema targetNamespace="http://www.ibm.com/wps/c2a">
            <xsd:simpleType name="WireStringType">
                <xsd:restriction base="xsd:string"></xsd:restriction>
            </xsd:simpleType>
        </xsd:schema>
    </types>

    <message name="MessageRequest">
        <part name="message_text" type="portlet:WireStringType" />
    </message>

    <portType name="ReceiveMessageService">
        <operation name="ReceiveMessageOperation">
            <input message="tns:MessageRequest" />
        </operation>
    </portType>

    <binding name="ReceiveMessageBinding" type="portlet:ReceiveMessageService">
        <portlet:binding />
        <operation name="ReceiveMessageOperation">
            <portlet:action name="ReceiveMessageAction" type="standard"
                actionNameParameter="ACTION_NAME" />
            <input>
              <portlet:param name="receive_text"
                partname="message_text" />
            </input>
        </operation>
    </binding>
```

Chapter 9. Developing RCP components  273
3. Save the changes.
4. Review the WSDL for this component and notice the WSDL file:
   - Defines a data type called WireStringType in the namespace of http://www.ibm.com/wps/c2a that has the restrictions of xsd:string.
   - Defines a message called MessageRequest that has a data item, message_text of type WireStringType.
   - Defines a port type name ReceiveMessageService that has an operation named ReceiveMessageOperation and contains an input message of type MessageRequest.
   - Declares a binding called ReceiveMessageBinding that implements the port type ReceiveMessageService. Here an action is declared named ReceiveMessageAction. For SWT components its type is standard. Its action name parameter is ACTION_NAME. This action declares the output data item named receive_text, that it can exchange.

**Note:** The property type and the namespace must be identical to the ones in the WSDL file for the source component. In this scenario the source portlet is the SendMessage portlet component.

**Update the WSDL file with output properties**

Although a WSDL file is not strictly necessary for an RCP component to send messages to the property broker, it is a best practice to declare all the properties the component is publishing and receiving. Besides, if the component is going to be wired using the Portal Tools for managed composite applications, the declaration of the SendMessage service becomes mandatory.

Follow these steps to update the SWTMessage.wsdl file:
1. Open the SWTMessage.wsdl file
2. Add the definitions shown in Example 9-11.

**Example 9-11  Adding publishing properties to the SWTMessage.wsdl file**

```xml
...<message name="MessageResponse">
    <part name="message_text" type="portlet:WireStringType" />
</message>

<portType name="SendMessageService">
    <operation name="SendMessageOperation">
        <output message="tns:MessageResponse" />
    </operation>
</portType>
```
3. Save the WSDL file.

**Updating the manifest.mf file**

Once you have the WSDL file in place you have to indicate which action is going to be executed when the a property is received by the sample SWT component. This is done by adding the extension, shown in Example 9-12, to the SWTMessage bundle’s plugin.xml file.

Follow these steps:

1. Open the MANIFEST.MF file for the com.ibm.itso.demoapp.swtmessage project, if it is not already opened.
2. Select the Extensions tab and click **Add**.
3. Uncheck **Show only extensions point from the required plug-ins**
4. Select `com.ibm.rcp.propertybroker.PropertyBrokerDefinitions` as shown in Figure 9-14.
5. Click Finish. Click No in the New plug-in dependency dialog.


7. Enter the details shown in Figure 9-15. Save the changes.
8. Select the plugin.xml tab and review the added handler definitions shown in Example 9-12.

Example 9-12 Mapping between the WSDL file and the action handler

```xml
...<extension
    point="com.ibm.rcp.propertybroker.PropertyBrokerDefinitions">
    <handler
        class="com.ibm.itso.demoapp.swtmessage.ActionHandler"
        file="wsdl/SWTMessage.wsd1"
        type="SWT_ACTION"/>
</extension>
...```

9. Review the following:
   a. The WSDL file is associated to an ActionHandler class that extends from the org.eclipse.jface.action.Action class.
   b. When a SWT component is receiving a property the type must be SWT_ACTION.
   c. The ActionHandler.runWithEvent() method will be called every time a message arrives to the SWTMessage component.
   d. The ActionHandler.runWithEvent() method is shown in Example 9-9 on page 271.

9.6.2 Adding wires to the composite application

Wires between components are declared in the plugin.xml file of the composite application. In this sample scenario, you will add two additional wires as required by the RCP component. See Figure 9-1 on page 252 for details.

Follow these steps to create the application wires:
2. Verify that there is an existing wire for the other two components in the composite application.
   Note: The property broker extension has been already created in the original composite application. For details on how to create the property broker extension see Chapter 7, “Developing portlet components” on page 153.
3. Right-click com.ibm.rcp.propertybroker.PropertyBrokerWire and select New -> wire. This is the wire to send messages from the SWTMessage
component to the ReceiveMessage portlet. Enter the details shown in Figure 9-16 and save the changes.

![Extension Element Details](image)

Figure 9-16  Wiring the SWTMessage component to the ReceiveMessage portlet

4. Right-click com.ibm.rcp.propertybroker.PropertyBrokerWire and select **New -> wire**. This is the wire to send messages from the SendMessage portlet to the SWTMessage component. Enter the details shown in Figure 9-17 and save the changes.
5. Select the plugin.xml and review the added wires. They should look as shown in Example 9-13.

Example 9-13  Added wires for the sample RCP component communications

```xml
<wire enable="true"
    sourceentityid="com.ibm.itso.demoapp.swtmessage.MainView:1"
    sourcename="send_text"
    targetentityid="/com.ibm.itso.demoapp.receivemessage.portlet/ReceiveMessagePortlet/default"
    targetname="MessageWireAction"
    targetparam="wire_text"
    type="PROPERTY_TO_ACTION"/>

<wire enable="true"
    sourceentityid="/com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet/default"
    sourcename="wire_text"
    targetentityid="com.ibm.itso.demoapp.swtmessage.MainView:1"
    targetname="ReceiveMessageAction"
    targetparam="receive_text"
    type="PROPERTY_TO_ACTION"/>

...
9.7 Running the composite application in Lotus Expeditor

Follow these steps to run the composite application with component wiring in the Lotus Expeditor Test Environment:

1. From the menu bar select Run -> Run. Select your Client Service configuration and click Run to run the Lotus Expeditor Test Environment.

2. Enter your keystore password or enter and confirm a new password if this is the first time you run the Lotus Expeditor client.

3. Select Open -> DemoAPP from the Lotus Expeditor menu to start the application.

4. Send a message from the SendMessage portlet, for example, enter a message in the SendMessage portlet and click Submit. You will see that the message is received by both the ReceiveMessage portlet component and the RCP SWTMessage component, as shown in Figure 9-18.

5. Now try sending a message from the SWTMessage component, for example enter a message in the SWTMessage component and click Send. You will see that the message is received by the ReceiveMessage portlet, as shown in Figure 9-19.
Figure 9-19  Sending a message from the SWTMessage component
Integrating AWT or Swing components in composite applications

This chapter provides a list of tasks and recommendations that you must complete to integrate an AWT or Swing application as a component of a composite application in Lotus Expeditor. A sample application is included to illustrate this process.

The following issues are explained in this chapter:

- Using the IBM Device Runtime Environment
- Integrating AWT/Swing applications as components using the SWT-AWT bridge
- AWT component intercommunication and wiring
10.1 AWT and Swing applications

AWT and Swing are the standard user interface APIs for the Java SE platform. As part of the Java SE platform they can be found in every machine with a standard Java Runtime Environment (JRE™).

The Abstract Windowing Toolkit (AWT) provides APIs for constructing user interface components such as menus, buttons, text fields, dialog boxes, check boxes, and for handling user input through those components. AWT features include:

- A small set of native user interface components
- An event-handling model
- Graphics and imaging tools, including shape, color, and font classes
- Layout managers, for window layouts that do not depend on a particular window size or screen resolution
- Data transfer classes, for cut-and-paste through the native platform clipboard

Swing is built on top of the AWT architecture. The Swing APIs also provide graphical component (GUI) for use in user interfaces. The Swing APIs are written in the Java programming language without any reliance on code that is specific to the GUI facilities provided by underlying operating system, so Swing is 100% pure Java. This allows the Swing GUI components to have a pluggable look-and-feel that can be switched while an application is running. The pluggable look and feel lets you create GUIs that can either look the same across platforms or can assume the look and feel of the current operating system platform (such as Microsoft® Windows, AIX® or Linux). Swing features include:

- GUI components, from buttons to complex tables
- Pluggable look and feel support
- Accessibility API
- Java 2D™ API, for to work with 2D graphics
- Drag and drop support
- Internationalization

Note: Swing was the project code name that built the new components. The official name for Swing is Java Foundation Classes.

More information about AWT/Swing can be found in the Java Tutorial available at:

http://java.sun.com/docs/books/tutorial/uiswing/index.html
10.2 SWT versus AWT

At the time when the Eclipse platform was being designed it became clear that the Java Swing user interface framework that comes integrated with Java would not allow market competitive solutions to be created. The original goal for Eclipse was to create competitive tooling platform for Windows and Linux. That goal has been achieved as this is largely down to the decision to build the low level user interface widgets so they map into the native operating system.

The Eclipse team also defined a clean interface between the native operating system and the Java API that would be implemented above it. This enables the SWT native widget code to be easily ported to other platforms. This was another major benefit of the Eclipse platform. Not only would it offers the developer performant user interfaces but it would offer a programming model and framework above the widgets that would ease the creation of rich and powerful applications.

There is often a debate about the merits of SWT versus Swing. The key is to remember why SWT was developed. First and foremost it was to support performant and competitive user interfaces on a native operating system. Swing on the other hand had its history in the Netscape Foundation Classes which was designed to support Rich internet applications delivered across the network.

If you wonder about the future of Swing, note that the new Java SE 6 platform has improved the AWT/Swing technology to the level that you hardly can differentiate a Swing component from a native one. There are also improvements on speed and efficiency.

The good news is that Swing user interface widgets can be easily integrated and work along side SWT widgets. This allows either panels or beans or just controls to be reused.

A more detailed comparison between AWT and SWT can be found in SWT, Swing or AWT: Which is right for you? at:


10.3 AWT components in Lotus Expeditor

Lotus Expeditor supports AWT/Swing components to build composite applications allowing you to reuse existing code. In most cases a standalone AWT application is packaged as a JAR file and its executed from the command line using the standard Java or javaw launcher. This means that a standalone
AWT application always has a main method, that acts as the application entry point.

When you integrate an existing AWT application in Lotus Expeditor, the application must be re-packaged as an OSGi bundle and the entry point is no more the application main() method. Instead the AWT application's main window (probably an AWT Frame, Swing JFrame or Swing JPanel) must be invoked from inside a SWT component, that in this case acts as a wrapper for the AWT application's main window.

Another consideration is that Lotus Expeditor comes with a fast and highly optimized JRE called jclDesktop, that does not include the libraries required to execute AWT/Swing components. So in order to integrate AWT/Swing components in your composite application you will need to install and use the full Java SE that for Lotus Expeditor comes as a separated component called the IBM Device Runtime Environment (DRE).

Once you have the IBM DRE in place, you need a way to run the AWT/Swing components on the Lotus Expeditor platform. Remember that Lotus Expeditor is an Eclipse-based product so what you will need is a way to run the AWT/Swing component inside a SWT component. Eclipse provides a class to perform this task called the SWT-AWT Bridge.

Because AWT and SWT are based on Java, you can invoke an AWT component's method from an inside a SWT component and vice versa, provided that the AWT component has some public methods to invoke. Additionally some considerations about threading should be observed as explained in “SWT-AWT bridge intercommunication” on page 288.

To summarize, in order to integrate an existing AWT application into an Lotus Expeditor composite application you have to:

- Repackage the AWT component as an OSGi bundle. This include to add the MANIFEST.MF file and the plugin.xml file; and a WSDL file in case Property Broker communication is required.
- Use the IBM DRE as the Java Runtime Environment for the Composite Application.
- Use the SWT_AWT Bridge to launch the AWT component main window inside a SWT composite.
- Make sure that you have observed the considerations about AWT-SWT intercommunication to avoid threading issues.
- Have access to the AWT component source code in case you need to modify it, for example to send a message using the Property Broker from the AWT
component or if you need the AWT component reacts to a message coming from another component through the Property Broker.

### 10.3.1 IBM Device Runtime Environment (DRE)

The IBM Device Runtime Environment provides Lotus Expeditor with a complete Java SE Runtime Environment in Linux and Windows platforms. IBM DRE comes bundled as a separated component to be installed on the Lotus Expeditor Toolkit or the Lotus Expeditor Client. For details about how to install IBM DRE see 10.4.1, “Installing the IBM Device Runtime Environment” on page 289.

### 10.3.2 The SWT-AWT bridge

The SWT_AWT bridge is a class that allows to run an AWT/Swing component embedded inside an SWT Composite and vice versa. This class has been available since Eclipse 3.0. Example 10-1 illustrates sample code showing how to run a Swing JPanel inside an SWT Composite.

**Example 10-1   Running a Swing JPanel inside a SWT Composite**

```java
import org.eclipse.swt.awt.SWT_AWT;
...
Composite composite = new Composite(parent, SWT.EMBEDDED);
java.awt.Frame frame = SWT_AWT.new_Frame(composite);
javax.swing.JPanel mainPanel = new javax.swing.JPanel();
frame.add(mainPanel);
...
```

The SWT_AWT.newFrame() method creates an AWT Frame container inside the SWT Composite. Note that the Composite container must be created using the SWT.EMBEDDED style.

**Important:** As of JDK™ 5.0, the embedded frame does not receive mouse events. Also when a lightweight component is added as a child of the embedded frame, the cursor does not change. In order to work around these problems, we strongly recommend that a heavyweight component, such as java.awt.Panel or javax.swing.JPanel, is added to the frame as the root of all components.

**Note:** Although of little or no value for Lotus Expeditor composite applications, you should also know that the SWT_AWT.newShell() method allows you to embed SWT components into an AWT container.
10.3.3 SWT-AWT bridge intercommunication

While displayed together, SWT and AWT run in different threads. To avoid deadlocks and other concurrency problems, you have to manage the intercommunication between SWT and AWT in a thread-safe manner.

To send messages from SWT to AWT you have to use the java.awt.EventQueue class. EventQueue is platform-independent class that queues events for AWT components. The EventQueue class has two methods of interest:

- `EventQueue.invokeLater(Runnable)` causes runnable to have its run method called in the dispatch thread of EventQueue. This will happen after all pending AWT events are processed.

- `EventQueue.invokeAndWait(Runnable)` is similar to invokeLater() but in this case the caller blocks until the run method is executed.

Example 10-2 illustrates how to send a message to AWT in a thread-safe way.

Example 10-2 Sending a message from SWT to AWT

```java
// We are inside the SWT thread, for example in the action listener
// of a SWT button. mainPanel is a javax.swing.JPanel instance.
...
java.awt.EventQueue.invokeLater(new Runnable() {
    public void run() {
        mainPanel.someMethod();
    }
});
```

Accordingly, to send messages from AWT to SWT you will use the SWT Display class. The `Display.asyncExec(Runnable)` method will execute the Runnable run method inside the SWT Display thread, as shown in Example 10-3.

Example 10-3 Sending a message from AWT to SWT

```java
// We are inside the AWT thread, for example in the action listener
// of an AWT button. messageLabel is an instance of the SWT Label
// widget
...
org.eclipse.swt.Display.getDefault().asyncExec(new Runnable() {
    public void run() {
```
10.4 Sample scenario

In this scenario you are going to extend the DemoApp application depicted in Chapter 7, “Developing portlet components” on page 153 with an existing AWT component.

10.4.1 Installing the IBM Device Runtime Environment

To develop or integrate AWT components you need to install the IBM DRE in your development environment. To install the IBM DRE in your development environment follow these steps:

1. Start your development environment, for example Rational Application Developer and Lotus Expeditor Toolkit.
2. From the menu bar, select Help → Software Updates → Find and install. Select Search for new features to install and click Next.
3. You have to add a new install location. Click Add Local Site.
4. Now you have to select the tooling update site location for the IBM DRE. If you expanded the images that come with Lotus Expeditor, it is located at <IBM_DRE_ROOT>/updates/tooling. Enter a meaningful name for the local site, for example IBM Device Runtime Environment as shown in Figure 10-1, and click OK.

![Edit Local Site](image)

Figure 10-1 Local site for the IBM Device Runtime Environment

5. Make sure that the newly added IBM DRE site is selected and click Finish.
6. The wizard will find the IBM DRE, as shown in Figure 10-2. Select **Lotus Expeditor Development Runtimes DRE** and click Next.

![Figure 10-2 Lotus Expeditor Development Runtime](image)

7. Accept the license agreement and click Next.

8. Make sure that you have enough disk space to install the runtime and click Finish to begin the installation.

9. If a warning about an unsigned feature appears, just ignore it and continue by clicking **Install All**.

10. Wait while the IBM DRE is installed. At the end, the wizard will ask you to restart the workbench. Click **Yes** to restart it.

Now you should have a new JRE available, you can look for it from the menu bar by selecting **Window → Preferences → Java → Installed JREs**. As shown in Figure 10-3 on page 291, there is a new JRE available called J2SE™ 5.0 Win32® x86. You can use this JRE to develop or integrate AWT/Swing components.
Chapter 10. Integrating AWT or Swing components in composite applications

10.4.2 The DemoApp application

The DemoApp application has two portlets that interchange messages between them using Property Broker via a wire. The source portlet sends messages through the wire, then the Property Broker does the matching and sends the message to the target portlet.

**Tip:** If you are not familiar with component intercommunication, refer to Chapter 5, “Component communication” on page 85.

The name for the source portlet is SendMessage and for the target portlet is ReceiveMessage. WSDL files are used by Lotus Expeditor to indicate what properties will be interchanged between portlet components. Example 10-4 shows the WSDL file for the SendMessage portlet.

---

**Figure 10-3** J2SE 5.0 Win32 X86 Runtime

**Note:** If the J2SE 5.0 Win32 x86 does not appear, reload the target features for Plugin Development and restart the workbench.
Example 10-4  SendMessage.wsdl

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="OrderDetail_Service"
  targetNamespace="http://www.ibm.com/wps/c2a"
  xmlns="http://schemas.xmlsoap.org/wsdl/
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.ibm.com/wps/c2a"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:portlet="http://www.ibm.com/wps/c2a">
<types>
<xsd:schema targetNamespace="http://www.ibm.com/wps/c2a">
  <xsd:simpleType name="WireStringType">
    <xsd:restriction base="xsd:string">
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
</types>
</definitions>
```

Example 10-5 shows the WSDL file for the ReceiveMessage portlet.
Example 10-5  ReceivingMessage.wsdl

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="OrderDetail_Service"
    targetNamespace="http://www.ibm.com/wps/c2a"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
    xmlns:tns="http://www.ibm.com/wps/c2a"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:portlet="http://www.ibm.com/wps/c2a">

<types>
    <xsd:schema targetNamespace="http://www.ibm.com/wps/c2a">
        <xsd:simpleType name="WireStringType">
            <xsd:restriction base="xsd:string"/>
        </xsd:simpleType>
    </xsd:schema>
</types>

<message name="WireRequest">
    <part name="wire_wsdl_text" type="portlet:WireReceivedStringType"/>
</message>

<portType name="Wire_Service">
    <operation name="WireOperation">
        <input message="tns:WireRequest"/>
    </operation>
</portType>

<binding name="ReceiveMessageBinding" type="portlet:Wire_Service">
    <portlet:binding/>
    <operation name="WireOperation">
        <portlet:action name="MessageWireAction"
            type="standard"
            actionNameParameter="ACTION_NAME"/>
        <input>
            <portlet:param name="wire_text" partname="wire_wsdl_text"/>
        </input>
    </operation>
</binding>
</definitions>
```
Note that the namespace http://www.ibm.com/wps/c2a and the property type WireStringType are the same in both WSDL files, this is a required for proper property sharing.

**Note:** The namespace and the property type should be the same in the WSDL files of other components participating in property broker communication.

### 10.4.3 The AWT Message component

The AWTMessage component is an existing standalone AWT application, in real life they probably come from a third-party vendor or part of the company's former or existing software. Example 10-6 shows the source code for the AWTMessage component.

#### Example 10-6  MainPanel.java

```java
package com.ibm.itso.awtsample.awtmessage;

import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;
import javax.swing.SwingConstants;

public class MainPanel extends JPanel {

    private static final long serialVersionUID = 1L;
    private JLabel jLabel = null;
    private JTextField message2SendText = null;
    private JLabel messageLabel = null;
    private JButton sendButton = null;

    public MainPanel() {
        super();
```

294  Building Composite Applications
private void initialize() {
    GridBagConstraints gridBagConstraints11 =
        new GridBagConstraints();
    gridBagConstraints11.gridy = 0;
    gridBagConstraints11.insets = new Insets(0, 5, 0, 5);
    gridBagConstraints11.gridx = 2;
    GridBagConstraints gridBagConstraints2 =
        new GridBagConstraints();
    gridBagConstraints2.gridy = 1;
    gridBagConstraints2.fill = GridBagConstraints.HORIZONTAL;
    gridBagConstraints2.gridwidth = 3;
    gridBagConstraints2.insets = new Insets(5, 5, 0, 5);
    gridBagConstraints2.gridx = 0;
    messageLabel = new JLabel();
    messageLabel.setText(" ");
    GridBagConstraints gridBagConstraints1 =
        new GridBagConstraints();
    gridBagConstraints1.fill = GridBagConstraints.HORIZONTAL;
    gridBagConstraints1.gridx = 1;
    gridBagConstraints1.weightx = 1.0;
    gridBagConstraints1.ipadx = 2;
    gridBagConstraints1.gridy = 0;
    GridBagConstraints gridBagConstraints = new GridBagConstraints();
    gridBagConstraints.gridy = 0;
    gridBagConstraints.ipadx = 1;
    gridBagConstraints.insets = new Insets(0, 5, 0, 5);
    gridBagConstraints.ipady = 0;
    gridBagConstraints.gridx = 0;
    JLabel = new JLabel();
    JLabel.setText("Message: ");
    JLabel.setHorizontalAlignment(SwingConstants.LEFT);
    this.setLayout(new GridBagLayout());
    this.setSize(300, 75);
    this.add(jLabel, gridBagConstraints);
    this.add(getMessage2SendText(), gridBagConstraints1);
    this.add(messageLabel, gridBagConstraints2);
    this.add(getSendButton(), gridBagConstraints11);
}

private JTextField getMessage2SendText() { 
    if (message2SendText == null) { 

    }
}
message2SendText = new JTextField();
message2SendText.setColumns(20);
return message2SendText;

private JButton getSendButton() {
    if (sendButton == null) {
        sendButton = new JButton();
        sendButton.setText("Send");
        sendButton.addActionListener(
            new java.awt.event.ActionListener() {
                public void actionPerformed(java.awt.event.ActionEvent e) {
                    sendMessage();
                }
            });
    }
    return sendButton;
}

protected void sendMessage() {

    if (getMessage2SendText().getText().equals(")") {
        JOptionPane.showMessageDialog(this, "Message cannot be empty",
            "Enter a message to send", JOptionPane.ERROR_MESSAGE);
        return;
    }

    final String message = getMessage2SendText().getText();
    messageLabel.setText("The message sent was "+ message + "!");
    getMessage2SendText().setText("");
}

public static void main(String[] args) {

    JFrame mainFrame = new JFrame("AWT Message Component");
    mainFrame.add(new MainPanel());
    mainFrame.setSize(400, 100);
    mainFrame.setVisible(true);
}
} // @jve:decl-index=0:visual-constraint="-356,-220"
The standalone AWT application can be tested using the AWTMessage.bat file located at the binary directory of the AWTMessageComponent.zip file in Appendix B, “Additional material” on page 721.

In order to integrate the AWTMessage component into the composite application the AWTMessage component has been repackaged in a separate bundle called com.ibm.itso.awtsample.awtmessage. Note that the bundle must use the IBM Device Runtime Environment instead the jclDesktop runtime. This is defined in the Java Build Path properties for the com.ibm.itso.awtsample.awtmessage project, as shown in Figure 10-4.

![Figure 10-4](image)

**Figure 10-4** Java Build Path properties for the AWT component project

Figure 10-5 shows how the AWTMessage component fits into the composite application. The AWTMessage component sends messages to the target portlet and receive messages from the source portlet.
The AWTMessage bundle is composed of a RCP View called MainView that contains a SWT Composite and an AWT Frame. The MainView view creates the SWT composite and attaches the composite to a new AWT Frame using the SWT_AWT bridge, then this frame is used as parent of a Swing JPanel instance called MainPanel as shown in Example 10-7.

Example 10-7 Creating the AWT and SWT containers

```java
... public class MainView extends ViewPart {
  ...
  private MainPanel mainPanel;
  ...
  public void createPartControl(Composite parent) {

    Composite c = new Composite(parent, SWT.EMBEDDED);
    Frame frame = SWT_AWT.new_Frame(c);
    mainPanel = new MainPanel();
    frame.add(mainPanel);
  }
...
The `MainPanel` class derives from `javax.swing.JPanel`. It has widgets on it to send and receive messages, as shown in Figure 10-6. The `messageLabel` widget shows the incoming messages from the `SendMessage` portlet. The `sendButton` is used to send the text written in the `message2SendText` widget to the `ReceiveMessage` portlet.

![Figure 10-6  MainPanel layout](image)

**Sending messages using the Property Broker**

When a user presses the Send button, the button's `actionListener()` method calls the `sendMessage` method shown in Example 10-8. The original `sendMessage` method has been extended to send a message through an RCP View using the Property Broker.

**Example 10-8  MainPanel.sendMessage() method**

```java
protected void sendMessage() {
    if (getMessage2SendText().getText().equals("") {  
        JOptionPane.showMessageDialog(this,
            "Message cannot be empty","Enter a message to send",
            JOptionPane.ERROR_MESSAGE);  
        return;
    }

    final String message = getMessage2SendText().getText();
    Display.getDefault().asyncExec(new Runnable() {
        public void run() {
            MainView view =
                (MainView)SWTHelper.locateView(MainView.ID);
            view.sendMessage(message);
```
The sendMessage() method performs the following tasks:
1. Perform some basic validation in the message2SendText widget to prevent sending empty messages.
2. Gets the message text from the message2SendText widget.
3. Uses the SWTHelper class to obtain a reference to the MainView.
4. Invokes the MainView.sendMessage() method.

Note that Step 3 and 4 are executed using the Display.asynExec() method, as was explained in “SWT-AWT bridge intercommunication” on page 288.

Example 10-9 shows the MainView.sendMessage() method.

Example 10-9  MainView.sendMessage() method

```java
... public void sendMessage(final String message) {
    PropertyController prop = PropertyFactory.createProperty();
    prop.setNamespace( Constants.PROPERTY_NAMESPACE );
    prop.setName( Constants.PROPERTY_NAME );
    prop.setType( Constants.PROPERTY_TYPE );
    PropertyValue propValue = PropertyFactory.createPropertyValue( prop, message );
    PropertyValue[] values = new PropertyValue[] { propValue };
    IViewPart view = SWTHelper.locateView(MainView.ID);
    try {
        SWTHelper.changedProperties(values, view);
    } catch (PropertyBrokerException e) {
        e.printStackTrace();
    }
    ...
The `MainView.sendMessage()` method performs these tasks:

1. Uses the `PropertyController` and `PropertyValue` classes to define the properties to exchange, a simple `String` in this case. The constants to create the properties are:
   - `PROPERTY_NAMESPACE`: `http://www.ibm.com/wps/c2a`
   - `PROPERTY_NAME`: `wire_text`
   - `PROPERTY_TYPE`: `WireStringType`

   Note that these values are the same that those defined in the WSDL file for the `ReceiveMessage` portlet, shown in Example 10-5 on page 293.

2. Uses the `SWTHelper.changedProperties()` method to publish these properties into the Property Broker. This method takes as parameters the view and the values to publish. The view will be used by the Property Broker to determine which components are going to receive the message. The link between the view and the target portlet is defined in the wire shown in Example 10-17 on page 306.

### Receiving messages from the Property Broker

So far you have seen how to publish a message from the `AWTMessage` component. The procedure to receive a message is slightly more complex. First of all the component has to define a WSDL file, declaring which properties the component will receive. In the `AWTMessage` bundle, the WSDL file is saved in the `/wsdl/AWTMessage.wsdl` file. The WSDL file for the `AWTMessage` component is shown in Example 10-10.

**Example 10-10  AWTMessage.wsdl file**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="OrderDetail_Service"
    targetNamespace="http://www.ibm.com/wps/c2a"
    xmlns="http://schemas.xmlsoap.org/wsdl/
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/
    xmlns:tns="http://www.ibm.com/wps/c2a"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:portlet="http://www.ibm.com/wps/c2a">
  <types>
    <xsd:schema targetNamespace="http://www.ibm.com/wps/c2a">
      <xsd:simpleType name="WireStringType">
        <xsd:restriction base="xsd:string">
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:schema>
  </types>
</definitions>
```
Note that both the property namespace and property type must be the same in the portlet and the AWT component.

Once you have the WSDL file in place, you have to indicate which action is going to be executed when a property is received by the AWTMessage component. This is done by adding the extension shown in Example 10-11 to the sample AWTMessage bundle’s plugin.xml file.

Example 10-11   Mapping between the WSDL file and the action handler

...<extension
  point="com.ibm.rcp.propertybroker.PropertyBrokerDefinitions">
  <handler
    class="com.ibm.itso.awtsample.awtmessage.ActionHandler"
    file="wsdl/AWTMessage.wsdl"
    type="SWT_ACTION"/>
</extension>
...


As you can see the WSDL file is associated to the ActionHandler class that extends from org.eclipse.jface.action.Action. When a SWT component is receiving a property the type must be SWT_ACTION. The ActionHandler.runWithEvent() method will be called each time a message arrives for the AWTMessage component. The ActionHandler.runWithEvent() method is shown in Example 10-12.

**Example 10-12  ActionHandler.runWithEvent()**

```java
...  
  public void runWithEvent(Event event) {

      if (event instanceof PropertyChangeEvent) {
          final PropertyChangeEvent finalEvent = (PropertyChangeEvent) event;

          Display.getDefault().asyncExec(new Runnable() {
              public void run() {

                  String message = (String) finalEvent.getPropertyValue().getValue();

                  String targetViewId = finalEvent.getWireDefinition().getTargetEntityId();

                  MainView mainView = (MainView) SWTHelper.locateView(targetViewId);

                  if (mainView != null) {
                      mainView.updateMessage(message);
                  }

              }
          });

      }  

  }
...  
```

The runWithEvent() method perform these tasks:

1. Cast the event object to PropertyChangeEvent. This class provides information about the message received.
2. Obtains the message from the event object.
3. Obtains the target view ID from the event object.
4. Obtains a reference to the target view using the `SWTHelper.locateView()` method.

5. Calls the `MainView.updateMessage()` method which in turn calls the `MainPanel.updateMessage()` method to update the `messageLabel` widget.

The link between the SendMessage portlet and the AWTMessage component is defined as a wire, shown in Example 10-17 on page 306.

Let us take a look to the `MainView.updateMessage()` method, shown in Example 10-13.

*Example 10-13  MainView.updateMessage() method*

```java
... 
   public void updateMessage(final String message) {
 
   java.awt.EventQueue.invokeLater(new Runnable() {
 
   public void run() {
 
   mainPanel.updateMessage(message);
 
   } });
 
   }

... 
```

Note that because the `MainView` class is invoking code in the `MainPanel`, the `EventQueue.invokeLater()` method is used as was explained in “SWT-AWT bridge intercommunication” on page 288.

Example 10-14 shows the `MainPanel.updateMessage()` method.

*Example 10-14  MainPanel.updateMessage() method*

```java
... 
   public void updateMessage(String message) {
 
   messageLabel.setText("The message received was " + message + ":");
 
   }

... 
```
10.4.4 The composite application

The composite application is assembled using a separate bundle representing the application itself. The bundle name is com.ibmitso.awtsample.application. This bundle contains the perspective classes, definitions for the application and definitions for the new wires and views.

Each portlet is displayed in the RCP Portlet Viewer that Lotus Expeditor provides. Example 10-15 shows how the portlets are declared in the application plugin.xml file in order to be displayed using the RCP Portlet Viewer.

Example 10-15 Declaration for displaying DemoApp portlets in the RCP viewer

```
...<extension point="com.ibm.rcp.portletviewer.portlets">
  <portletData
    contextroot="/com.ibm.itso.demoapp.sendmessage.portlet"
    entityId="com.ibm.itso.demoapp.sendmessage.portlet"
    portletname="SendMessagePortlet"
    portletwindowid="default"
    title="Send Message"/>

  <portletData
    contextroot="/com.ibm.itso.demoapp.receivemessage.portlet"
    entityId="com.ibm.itso.demoapp.receivemessage.portlet"
    portletname="ReceiveMessagePortlet"
    portletwindowid="default"
    title="Receive Message"/>

</extension>
...```

The application components are layout using an Eclipse perspective. Example 10-16 shows how the components are layout using the AWTSamplePerspectiveFactory perspective class.

Example 10-16 Layout of components using a perspective

```
...public void createInitialLayout(IPageLayout pageLayout) {

  pageLayout.addStandaloneView(SEND_MESSAGE_VIEWER_ID, true,
    IPageLayout.TOP, 0.45f, pageLayout.getEditorArea());

...```
The wires between components are also declared in the application plugin.xml file, as shown in Example 10-17.

Example 10-17  Wires for component communication

...<extension point="com.ibm.rcp.propertybroker.PropertyBrokerWire">

<wire enable="true"
    sourceentityid="/com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet/default"
    sourcename="wire_text"
    targetentityid="com.ibm.itso.awtsample.awtmessage.mainview:1"
    targetname="ReceiveMessageAction"
    targetparam="wire_text"
    type="PROPERTY_TO_ACTION"/>

<wire enable="true"
    sourceentityid="/com.ibm.itso.demoapp.receivemessage.portlet/ReceiveMessagePortlet/default"
    sourcename="wire_text"
    targetentityid="/com.ibm.itso.demoapp.receivemessage.portlet/ReceiveMessagePortlet/default"
    targetname="MessageWireAction"
    targetparam="wire_text"
    type="PROPERTY_TO_ACTION"/>

<wire enable="true"
    sourceentityid="/com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet/default"
    sourcename="wire_text"
    targetentityid="/com.ibm.itso.demoapp.receivemessage.portlet/ReceiveMessagePortlet/default"
    targetname="MessageWireAction"
10.4.5 Configuring the test environment for running the application

By default Lotus Expeditor test environment runs using the jclDesktop runtime. In order to run the sample application you have to change the runtime to J2SE 5.0. From the menu bar of your development environment, select Run -> Run, select your Client Service configuration, and make sure that Runtime JRE is set to J2SE 5.0 Win32 x86, as shown in Figure 10-7.

![Figure 10-7 Using J2SE 5.0 as the runtime when launching the application](image)

10.4.6 Summary

Here is a summary of the tasks to be done in order to create the sample application:
1. Install the IBM Device Runtime Environment in your development environment. Refer to “Installing the IBM Device Runtime Environment” on page 289 for more information.


3. Create a new Client Services project called com.ibm.itso.awtsample.awtmessage. This bundle contains the AWTMessage component.

4. Change the JRE for the com.ibm.itso.awtsample.awtmessage project to J2SE 5.0 Win32 x86 in order to have the necessary AWT classes in place.

5. Copy source code from the original AWTMessage component into the src directory in the com.ibm.itso.awtsample.awtmessage bundle. You can copy the code from the source directory located as described in Appendix B, “Additional material” on page 721.

6. Create a view inside the com.ibm.itso.awtsample.awtmessage bundle called MainView on the com.ibm.itso.awtsample.awtmessage package. This view creates the SWT composite, the AWT Frame and the MainPanel instance. You can copy the code for the MainView class from Additional Materials, as described in Appendix B, “Additional material” on page 721.

7. Add a entry for the MainView view to the awtmessages bundle’s plugin.xml file.

8. Modify the MainPanel class to send a message to the Property Broker using the MainView class. You have to modify the MainPanel’s sendMessage() method as shown in Example 10-8 on page 299.

9. Create the ReceiveMessage WSDL file and place it in the /wsdl/ directory. You can copy the file from Additional Materials, see Appendix B, “Additional material” on page 721.

10. Create the ActionHandler class to manage the incoming messages. You can copy the code for the ActionHandler class from Additional Materials, see Appendix B, “Additional material” on page 721.

11. Add an entry to the awtmessages bundle’s plugin.xml file to map the ReceiveMessage WSDL file to the ActionHandler class, as shown in Example 10-11 on page 302.

12. Create a new Client Services project called com.ibmitso.awtsample.application. This bundle contains the application definition, the wire declarations and the perspective class.

13. Create a new PerspectiveFactory class called AWTSamplePerspectiveFactory. You can copy the code for the
AWTSamplePerspectiveFactory class from Additional Materials, see Appendix B, “Additional material” on page 721.

14. Add the application definition to the application bundle’s plugin.xml.

15. Add the portlets declaration to the application bundle’s plugin.xml as shown in Example 10-15 on page 305.

16. Add the wires declaration to the application bundle’s plugin.xml as shown in Example 10-17 on page 306.

17. Modify the Lotus Expeditor test environment to use the J2SE 5.0 Win32 x86 runtime, as shown in “Configuring the test environment for running the application” on page 307.

18. Launch the application by using the Run -> Run command from the menu bar.
Debugging the wired composite applications in Lotus Expeditor

When developing composite applications that use wires for component intercommunication, eventually you will need to perform some type of debugging work. For example, when, for some reason, messages sent by a component are not arriving to the specified target.

Lotus Expeditor Client provides a mechanism to query the Property Broker for declared communication artifacts, such as actions and properties. Lotus Expeditor Client also provides a way to trace the flow of selected properties from the source component to the target component. These mechanisms are provided at the OSGi level through the use of commands issued using the integrated Lotus Expeditor Client OSGi console.

This chapter illustrates sample execution of the most important commands used to troubleshoot problems in the property broker.
11.1 Working with the OSGi console

The OSGi console is available with the Lotus Expeditor Client Toolkit in the Console view of your Eclipse-based development environment when you start the test environment. Figure 11-1 illustrates an example of the OSGi console in Rational Application Developer.

![Figure 11-1  OSGi console in Rational Application Developer](image)

In order to work with the OSGi console when using the Lotus Expeditor Client runtime you need to add the -console parameter to the Lotus Expeditor shortcut. For example, Figure 11-2 shows how to add this parameter in a Windows XP environment.
Chapter 11. Debugging the wired composite applications in Lotus Expeditor

The OSGi console for the Lotus Expeditor Client runtime running on Windows XP is shown in Figure 11-3.

Figure 11-2 Modifying the Expeditor Client launcher to show the console

Figure 11-3 OSGi console in Expeditor Client runtime
11.2 Property Broker commands

In this section, the most important commands are executed to show declarative information related to wires and actions. These commands are executed when running the DemoApp sample composite application developed in Chapter 7, “Developing portlet components” on page 153.

For example:

- `pbsh a`. This command shows all actions.

Example 11-1  Showing all actions

```
osgi> pbsh a
------------------------------------
Property Broker All Actions
------------------------------------
NAME: DummyGetURLAction
  Title: notifies users when the URL is changed
  Description: notifies users when the URL is changed
  Handler Type: COMMAND
  Runnable Type:
    com.ibm.rcp.ui.internal.browser.portal.actions.EBAction
  Owner ID: com.ibm.rcp.ui.browser.portal
  Name Parameter: ACTION_NAME
  Parameters: 1 parameters
    Title = Current URL changes property
    Description =
    Property = OutURL
    Property Title = Current URL changes property
    Property NS = http://www.ibm.com/wps/eb
    Property ClassName = java.lang.String
    Property default Value = null
    Direction: [OUT]
```

```
NAME: MessageWireAction
  Title:
  Description:
  Handler Type: portlet
  Runnable Type: javax.portlet.PortletSessionUtil
  Owner ID:
    /com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet
  Name Parameter: ACTION_NAME
  Parameters: 1 parameters
    Title =
```
**Example 11-2 Showing all active actions**

```
> pbsh aa. This command shows all active actions.

osgi> pbsh aa
```

Property Broker Active Actions

```
NAME: DummyGetURLAction
  Title: notifies users when the URL is changed
  Description: notifies users when the URL is changed
  Handler Type: COMMAND
  Runnable Type: com.ibm.rcp.ui.internal.browser.portal.actions.EBAction
  Owner ID: com.ibm.rcp.ui.browser.portal
  Name Parameter: ACTION_NAME
  Parameters: 1 parameters
```
Title = Current URL changes property
Description = 
Property = OutURL
Property Title = Current URL changes property
Property NS = http://www.ibm.com/wps/eb
Property ClassName = java.lang.String
Property default Value = null
Direction: [OUT]

NAME: MessageWireAction
Title: 
Description: 
Handler Type: portlet
Runnable Type: javax.portlet.PortletSessionUtil
Owner ID: /com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet
Name Parameter: ACTION_NAME
Parameters: 1 parameters
Title = 
Description = 
Property = wire_text
Property Title = 
Property NS = http://www.ibm.com/wps/c2a
Property ClassName = java.lang.String
Property default Value = null
Direction: [OUT]

NAME: MessageWireAction
Title: 
Description: 
Handler Type: portlet
Runnable Type: javax.portlet.PortletSessionUtil
Owner ID: /com.ibm.itso.demoapp.receivemessage.portlet/ReceiveMessagePortlet
Name Parameter: ACTION_NAME
Parameters: 1 parameters
Title = 
Description = 
Property = wire_text
Property Title = 
Property NS = http://www.ibm.com/wps/c2a
Property ClassName = java.lang.String
Property default Value = null
Direction: [IN]

▶ **pbsh p.** This command shows all properties by owner.

*Example 11-3  Showing all properties by owner*

```text
osgi> pbsh p
-----------------------------------------
Property Broker Properties
-----------------------------------------
Owner = /com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet
There are 1 properties registered.
-----------------------------------------
Name: wire_text
Namespace: http://www.ibm.com/wps/c2a
Description:
Title:
Type: WireStringType
Class: class
    com.ibm.rcp.propertybroker.internal.property.PropertyImpl
Default: null
Direction: [OUT]
Is Wired: YES
-----------------------------------------
Owner = com.ibm.rcp.ui.browser.portal
There are 9 properties registered.
-----------------------------------------
Name: Window
Namespace: http://www.ibm.com/wps/eb
Description:
Title: New browser view is opened property
Type: OpenWindowType
Class: class
    com.ibm.rcp.propertybroker.internal.property.PropertyImpl
Default: null
Direction: [OUT]
Is Wired: NO
```

```
...  `-----------------------------------------
Owner = /com.ibm.itso.demoapp.receivemessage.portlet/ReceiveMessagePortlet
There are 1 properties registered.
-----------------------------------------
Name: wire_text
```
 Namespace: http://www.ibm.com/wps/c2a
  Description:
  Title: 
  Type: WireStringType 
  Class: class 
       com.ibm.rcp.propertybroker.internal.property.PropertyImpl 
  Default: null
  Direction: [IN]
  Is Wired: YES

-----------------------------------------

 pbsh p <owner>. This command shows all properties for this owner.

Example 11-4  Showing all properties for a designated owner

osgi> pbsh p
    /com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet

-----------------------------------------

Property Broker Properties By Owner
OWNER: /com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet

-----------------------------------------

Name: wire_text 
   NameSpace: http://www.ibm.com/wps/c2a 
   Description:
   Title: 
   Type: WireStringType 
   Class: class 
       com.ibm.rcp.propertybroker.internal.property.PropertyImpl 
   Default Value: null
   Is Wired: YES

-----------------------------------------

 pbsh w. This command shows all enabled wires.

Example 11-5  Showing all enabled wires

osgi> pbsh w

-----------------------------------------

Property Broker Enabled Wires

-----------------------------------------

Title: null
Id: 
    Owner Id: com.ibm.demoapp.application
Ordinal: -1
Type: PROPERTY_TO_ACTION_WIRE
Source Name: wire_text
Source Entity ID:
/com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet/default
Source Param: null
Target Name: MessageWireAction
Target Entity ID:
/com.ibm.itso.demoapp.receivemessage.portlet/ReceiveMessagePortlet/default
Target Param: wire_text

---

▶ pbsh aw. This command shows all wires.

Example 11-6  Showing all wires

osgi> pbsh aw

Property Broker All Wires

Owner = com.ibm.demoapp.application
There are 1 wires registered.

Title: null
Id:
Owner Id: com.ibm.demoapp.application
Ordinal: -1
Type: PROPERTY_TO_ACTION_WIRE
Source Name: wire_text
Source Entity ID:
/com.ibm.itso.demoapp.sendmessage.portlet/SendMessagePortlet/default
Source Param: null
Target Name: MessageWireAction
Target Entity ID:
/com.ibm.itso.demoapp.receivemessage.portlet/ReceiveMessagePortlet/default
Target Param: wire_text

---

▶ pbsh ns. This command shows the registered name spaces.
Example 11-7  Showing all registered namespaces

```
osg> pbsl ns
-----------------------------------------
Property Broker Registered NameSpaces
-----------------------------------------
Registered Namespaces are 2
-----------------------------------------
  1. "http://www.ibm.com/wps/eb" has 9 properties registered
  2. "http://www.ibm.com/wps/c2a" has 1 properties registered
-----------------------------------------
```

There are commands that allow you to see the path of a changed property during runtime. You do not have to enable logging; the messages display as warnings so they appear in the default log.

► **pbt <Property>**. This command traces the path for the specified property.

Example 11-8  Enabling trace for a property

```
osgi> pbt wire_text
Property broker is now tracing the property "wire_text"
```

► **pblt**. This command displays a list of the currently traced properties.

Example 11-9  Listing all traced properties

```
osgi> pblt
-----------------------------------------
Registered Property Traces are:
-----------------------------------------
  1. wire_text
```

► **pbut <Property>**. This command stops traces for the specified property, it removes the trace.

Example 11-10  Disabling trace for a property

```
osgi> pbut wire_text
Property broker has stopped tracing "wire_text"
```

If you enable tracing for a property, a log message is displayed when the property comes to the Property Broker as shown in Example 11-11.

Example 11-11  Sample log output for a traced property

```
2007/03/26 21:22:53.484 WARNING PBTRACE(wire_text) changedProperties was called.
```
::class.method=com.ibm.rcp.propertybroker.internal.PropertyBrokerDispat
er.changedProperties() ::thread=HttpServer : 2
::loggername=com.ibm.rcp.propertybroker
2007/03/26 21:22:53.578 WARNING PBTRACE(wire_text) Testing Property
with Wire
(PROPERTY_TO_ACTION_WIRE:wire_text:MessageWireAction:/com.ibm.itso.demo
app.sendmessage.portlet/SendMessagePortlet/default:/com.ibm.itso.demoap
p.receivemessage.portlet/ReceiveMessagePortlet/default)
::class.method=com.ibm.rcp.propertybroker.internal.PropertyBrokerDispat
er.changedProperties() ::thread=HttpServer : 2
::loggername=com.ibm.rcp.propertybroker
2007/03/26 21:22:53.593 WARNING PBTRACE(wire_text) is being added to
the queue.
::class.method=com.ibm.rcp.propertybroker.internal.PropertyBrokerDispat
er.doPropertyToAction() ::thread=HttpServer : 2
::loggername=com.ibm.rcp.propertybroker
2007/03/26 21:22:53.593 WARNING PBTRACE(wire_text) Adding property to Q
(count = 0)
::class.method=com.ibm.rcp.propertybroker.internal.PropertyBrokerDispat
er.addPropertyToAction() ::thread=HttpServer : 2
::loggername=com.ibm.rcp.propertybroker
2007/03/26 21:22:53.609 WARNING PBTRACE(wire_text) Added property to Q
(count = 1)
::class.method=com.ibm.rcp.propertybroker.internal.PropertyBrokerDispat
er.addPropertyToAction() ::thread=HttpServer : 2
::loggername=com.ibm.rcp.propertybroker
2007/03/26 21:22:53.609 WARNING PBTRACE(<all>) Calling notify to
process Q
::class.method=com.ibm.rcp.propertybroker.internal.PropertyBrokerDispat
er.completeAdd() ::thread=HttpServer : 2
::loggername=com.ibm.rcp.propertybroker
2007/03/26 21:22:53.625 WARNING PBTRACE(<all>) Processing Q (count = 1)
::class.method=com.ibm.rcp.propertybroker.internal.PropertyBrokerDispat
er.run() ::thread=Worker-1 ::loggername=com.ibm.rcp.propertybroker
2007/03/26 21:22:53.640 WARNING PBTRACE(wire_text) calling handler
(com.ibm.rcp.propertybroker.portlet.internal.PortletHandler) for Action
(MessageWireAction)
::class.method=com.ibm.rcp.propertybroker.internal.PropertyBrokerDispat
er.qCallAction() ::thread=Worker-1
::loggername=com.ibm.rcp.propertybroker
2007/03/26 21:22:53.656 WARNING Portlet handler received property
change for wire_text
::class.method=com.ibm.rcp.propertybroker.portlet.internal.PortletHandl
er.forwardEventToPortlet() ::thread=Worker-1
::loggername=com.ibm.rcp.propertybroker.portlet
2007/03/26 21:22:53.984 WARNING Property change (wire_text) sent to portlet.
::class.method=com.ibm.rcp.propertybroker.portlet.internal.PortletHandler.forwardEventToPortlet() ::thread=Worker-1
::loggername=com.ibm.rcp.propertybroker.portlet
Chapter 12. Integrating OLE documents and ActiveX Controls in composite applications

This chapter describes a way to integrate OLE Documents and ActiveX Controls (COM objects) into Lotus Expeditor on Windows platforms. In particular, we show you how to use the OLE interface to integrate not only applications but documents, such as Word and Excel® as components in composite applications.

The OLE interface allows applications to communicate and share properties without having to know about each other.

Note: For details about how to integrate OLE Documents and ActiveX Controls into an application using SWT, see ActiveX Support In SWT by Veronika Irvine, OTI, at the following site:

http://www.eclipse.org/articles/Article-ActivexSupportInSwt/index.html
12.1 Overview

Object Linking and Embedding (OLE) is a technology that allows you to build compound documents. A compound document is something that can display any content that supports the OLE specification, for example you can create a Word document with an embedded Excel spreadsheet.

OLE is part of the more comprehensive ActiveX technology. ActiveX is a set of object-oriented programming technologies and tools. With ActiveX you can create components that runs in Windows environments. These components are called ActiveX controls. Note that many programs widely used in Windows environments are also ActiveX controls. For example, the Microsoft Internet Explorer® browser provides an ActiveX interface to interact with.

Lotus Expeditor inherits its OLE and ActiveX support from Eclipse. OLE Objects can be contained in SWT widgets. From there, they can be activated and deactivated for user interaction, and manipulated by the application according to their specification. For example, edited data and state can be retrieved or saved. If the object is no longer required, it can be disposed.

An OLE Document or ActiveX Control is added to a composite application component by inserting it into a container. The container is made up of two parts:

- The *OleFrame* object handles sizing, menu management and window placement.
- The *OleClientSite* used for OLE Documents or the *OleControlSite* used for ActiveX Controls. They handle interactions with a specific OLE Object.

*Note:* These classes belong to the org.eclipse.swt.ole.win32 package. You will need to include this package in the classes using OLE objects.

12.2 Steps to integrate an OLE object as a component of a composite application

You will need to do the following when integrating an OLE object as a component of a composite application in Lotus Expeditor:

1. Create the OleFrame. See Example 12-1.

   Example 12-1 Creating an OLE frame
   
   ```java
   OleFrame frame = new OleFrame(composite, SWT.NONE);
   ```
2. Next, you create either an OleClientSite or an OleControlSite. For example:
   a. Create an OleClientSite for a storage file. The storage file has an OLE format with information about the type of OLE Object that can view it. For example, a ".doc" file created by Word is a Storage file. Given a Storage file, OLE will figure out which OLE Document to create. Example 12-2 illustrates and example to create a OleClientSite linked to a word document.

   **Example 12-2 Creating an OleClientSite**

   ```java
   File file = new File("C:\OleDocumentation.doc");
   OleClientSite clientSite = new OleClientSite(frame, SWT.NONE, file);
   ```

   b. Alternatively, create an OleControlSite for the ActiveX Control. For example, the ProgramID for the Internet Explorer is `Shell.Explorer`. The Web browser can be embedded in an application. Example 12-3 illustrates sample code to accomplish this.

   **Example 12-3 Creating an OleControlSite**

   ```java
   OleControlSite controlSite = new OleControlSite(frame, SWT.NONE, "Shell.Explorer");
   ```

3. Activate the OLE Object as in-place activation. This is done by invoking the `doVerb` action on the OleClientSite or on the OleControlSite. The `doVerb` syntax is shown in Example 12-4.

   **Example 12-4 Syntax for the doVerb() method**

   ```java
   public int doVerb(int verb)
   ```

**Verbs**
Verb is an integer value mapped to one of the predefined verb shown in Table 12-1.
Table 12-1  Verb values

<table>
<thead>
<tr>
<th>Verb</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLE.OLEIVERB_PRIMARY</td>
<td>Specifies the action when the end user double-clicks the object.</td>
</tr>
<tr>
<td>OLE.OLEIVERB_SHOW</td>
<td>Shows the object.</td>
</tr>
<tr>
<td>OLE.OLEIVERB_OPEN</td>
<td>Opens the object in a separate window.</td>
</tr>
<tr>
<td>OLE.OLEIVERB_HIDE</td>
<td>Removes the object user interface from the view.</td>
</tr>
<tr>
<td>OLE.OLEIVERB_INPLACEACTIVATE</td>
<td>Activates an object in place without displaying tools, such as menus and toolbars.</td>
</tr>
<tr>
<td>OLE.OLEIVERB_UIACTIVATE</td>
<td>Activates an object in place with its full set of user-interface tools.</td>
</tr>
<tr>
<td>OLE.OLEIVERB_DISCARDUNDOSTATE</td>
<td>Discards any undo state.</td>
</tr>
</tbody>
</table>

Note: If the doVerb command is successful, the return value is OLE.S_OK.

IDispatch interface

In order to manipulate an OLE document or an ActiveX control you can use the IDispatch interface. IDispatch provides access to get and set property values and invoke methods. For example, Word provides the entire Word Basic interface that gives you access to all sorts of commands and properties like cut/copy/paste, print, spell check, select text, change paragraph format, etc. In SWT the IDispatch capabilities are accessed using the OleAutomation object.

OleAutomation object

An OleAutomation object can be created from a client or control site or it can be obtained as the return value from a method invocation. For example the Web Browser provides commands like navigate, back, forward, home which you can access as shown in Example 12-5.

Example 12-5  Using OleAutomation to access OLE properties and actions

```java
...  OleControlSite controlSite = new OleControlSite(frame, SWT.NONE, "Shell.Explorer");  OleAutomation automation = new OleAutomation(controlSite);  int[] rgdispid = automation.getIDsOfNames(new String[] { "Navigate" });
```
int dispIdMember = rgdispid[0];
Variant[] rgvarg = new Variant[1]; // this is the URL parameter
rgvarg[0] = new Variant("www.ibm.com");
Variant pVarResult = automation.invoke(dispIdMember, rgvarg);

Note: The return type is a Variant containing the value of the property. A Variant is a generic way of passing around data. You can ask the Variant for the data in the format that you prefer.

For more information about OLE and ActiveX, refer to the MSDN® site at:

### 12.3 Sample scenario

In the sample scenario you will see how to use the classes depicted before to include an OLE document and an ActiveX control. The sample composite application, including two native components, is illustrated in Figure 12-1.

![Sample composite application with native components](image)

Figure 12-1 Sample composite application with native components

Follow these steps:
1. Start your Lotus Expeditor development environment. Note that this scenario will work only in Windows platforms.

2. Create a new Client Services project called
   com.ibm.itso.nativeapp.olesample.

3. Create a view named OLEDocumentView. Replace the generated code with the example shown in Example 12-6. This view shows a standard Open File dialog to select a Word document that in turn is shown embedded inside the view.

Example 12-6   OLEDocumentView.java

```java
package com.ibm.itso.nativeapp.olesample;

import java.io.File;
import org.eclipse.swt.SWT;
import org.eclipse.swt.ole.win32.OLE;
import org.eclipse.swt.ole.win32.OleClientSite;
import org.eclipse.swt.ole.win32.OleFrame;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.swt.widgets.FileDialog;
import org.eclipse.ui.part.ViewPart;

public class OLEDocumentView extends ViewPart {

    public static final String VIEW_ID =
        "com.ibm.itso.nativeapp.olesample.view2";

    public OLEDocumentView() {
        // TODO Auto-generated constructor stub
    }

    public void createPartControl(Composite composite) {

        OleFrame frame = new OleFrame(composite, SWT.NONE);

        FileDialog fd = new FileDialog(composite.getShell(), SWT.OPEN);
        fd.setText("Open");
        fd.setFilterPath("C:/$");
        String[] filterExt = {"*.doc", ".xls"};
        fd.setFilterExtensions(filterExt);
        String selected = fd.open();

        File file = new File(selected);
        OleClientSite clientSite =
```
new OleClientSite(frame, SWT.NONE, file);

    clientSite.doVerb(OLE.OLEIVERB_SHOW);
}

public void setFocus() {
// TODO Auto-generated method stub
}


4. Create a view named OLEBrowserView. Replace the generated code with the example shown in Example 12-7. This view shows the Internet Explorer browser invoked through its OLE interface. Note how the ActiveX control is manipulated to show a specific URL.

Example 12-7  OLEBrowserView.java

package com.ibm.itso.nativeapp.olesample;

import org.eclipse.swt.SWT;
import org.eclipse.swt.ole.win32.OLE;
import org.eclipse.swt.ole.win32.OleAutomation;
import org.eclipse.swt.ole.win32.OleControlSite;
import org.eclipse.swt.ole.win32.OleFrame;
import org.eclipse.swt.ole.win32.Variant;
import org.eclipse.swt.widgets.Composite;
import org.eclipse.ui.part.ViewPart;

public class OLEBrowserView extends ViewPart {

    public static final String VIEW_ID =
    "com.ibm.itso.nativeapp.olesample.view1";

    public OLEBrowserView() {
        // TODO Auto-generated constructor stub
    }

    public void createPartControl(Composite composite) {
        OleFrame frame = new OleFrame(composite, SWT.NONE);

        OleControlSite controlSite = new OleControlSite(frame, SWT.NONE,
"Shell.Explorer");
controlSite.doVerb(OLE.OLEIVERB_SHOW);

OleAutomation automation = new OleAutomation(controlSite);
int[] rgdispid = automation.getIdsOfNames(
    new String[] { "Navigate" });
int dispIdMember = rgdispid[0];
Variant[] rgvarg = new Variant[1]; // this is the URL parameter
rgvarg[0] = new Variant("www.ibm.com");
Variant pVarResult = automation.invoke(dispIdMember, rgvarg);
System.out.println(pVarResult);

}

public void setFocus() {
    // TODO Auto-generated method stub
}

5. Create a perspective named PerspectiveFactory. Replace the generated code with the one shown in Example 12-8. The perspective lays out the previously created views.

Example 12-8  PerspectiveFactory

package com.ibm.itso.nativeapp.olesample;

import org.eclipse.ui.IPageLayout;
import org.eclipse.ui.IPerspectiveFactory;

public class PerspectiveFactory implements IPerspectiveFactory {

    public void createInitialLayout(IPageLayout pageLayout) {

        pageLayout.addStandaloneView(
            OLEBrowserView.VIEW_ID, true, IPageLayout.LEFT,
            0.45f, pageLayout.getEditorArea());

        pageLayout.addStandaloneView(
            OLEDocumentView.VIEW_ID, true, IPageLayout.RIGHT,
            0.45f, OLEBrowserView.VIEW_ID);

        pageLayout.setEditableAreaVisible(false);

    }

}
6. Finally add the declarations for the views, the perspective and the application itself to the plugin.xml file, as shown in Example 12-9.

Example 12-9  Sample plugin.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension point="org.eclipse.ui.perspectives">
    <perspective
      class="com.ibm.itso.nativeapp.olesample.PerspectiveFactory"
      id="com.ibm.itso.nativeapp.olesample.perspective"
      name="OLE Sample"/>
  </extension>
  <extension point="org.eclipse.ui.views">
    <view
      class="com.ibm.itso.nativeapp.olesample.OLEBrowserView"
      id="com.ibm.itso.nativeapp.olesample.view1"
      name="OLE Browser"/>
    <view
      class="com.ibm.itso.nativeapp.olesample.OLEDocumentView"
      id="com.ibm.itso.nativeapp.olesample.view2"
      name="OLE Document"/>
  </extension>
    <Application
      DisplayName="OLE Sample"
      PerspectiveId="com.ibm.itso.nativeapp.olesample.perspective"/>
  </extension>
</plugin>
```

7. Run the application using the Run → Run option from the menu bar. Create a Client Services configuration if necessary.

8. Open the application selecting OLE Sample from the Open menu. Choose a Word document to open. The application should look similar as illustrated in Figure 12-2.
Figure 12-2  OLE sample composite application

OLE and Activex Support

When an OLE (Object Linking and Embedding) object is used in a composite application, the OLE object must be activated. This is typically done by double-clicking on the OLE object in the composite application.

OLE7SAMPLE: A sample OLE application that demonstrates the activation of OLE objects. It includes a window that displays information about the OLE objects in the application.

OLE7SAMPLE.XML: An XML file that defines the structure of the OLE objects in the application.
Using the OSGi Event Admin

This chapter describes how you can develop composite applications that run on Lotus Expeditor using the OSGi Event Admin mechanism, in order to archive communication among its different components.

In this chapter, you will enhance an existing application (ITSO Car Rental) by implementing code that allows a Web view to communicate with an RCP view using OSGi technology.
13.1 Overview

Lotus Expeditor is built on the Eclipse Rich Client Platform, which includes an Open Services Gateway initiative (OSGi) framework. The framework is based upon the OSGi Service Platform Release 4 specification with additional extensions provided by the Eclipse 3.2.1 implementation of the OSGi framework.

Application developers partition applications into services and other resources. Services and resources are packaged into bundles, which are files that serve as the delivery unit for applications. Bundles have manifests with special headers that enable you to share classes and services at the package level. Within the Eclipse based platforms, all plug-ins are OSGi bundles, so you can think of the terms plug-in and bundle as being interchangeable.

**OSGi Event Admin**

Nearly all the bundles in an OSGi framework must deal with events, either as an event publisher or as an event handler. The Event Admin service provides an inter-bundle communication mechanism. It is based on an event publish and subscribe model, popular in many message based systems.

13.2 ITSO Car Rental scenario

In this scenario, you will enhance an existing composite application, the ITSO Car Rental application, to achieve intercommunication between a Web view and an RCP view using the OSGi Event Admin API.

The ITSO Car Rental application consists of four views:

- An RCP view which can be used to search accounts through a customer number search function, which refreshes the view with the customer information.
- A Portlet view which contains payment information about the selected customer.
- A Portlet view which shows a list of available car models available for rental.
- A Web view which accepts the entry of a Customer Number and shows the cars rented by the selected customer.

Figure 13-1 provides a screen capture of the existing ITSO Car Rental application.
Chapter 13. Using the OSGi Event Admin

Enhancing the application

Currently, when a customer number is entered in the Web view the customer information is only updated on that view. In this chapter, you will add code that uses the OSGi Event Admin to send the customer number from the Web view to the RCP view so that the information can be automatically refreshed.

13.2.1 Using the OSGi Event Admin

The instructions provided below take you through the steps necessary to update the ITSO Car Rental application to use the OSGi Event Admin in order to achieve communication between its Web view and its RCP view. In the steps to follow, note that a significant portion of the programmatic code is written for you by the Rational Application Developer tooling wizards and thus significantly speeds-up development time.

During this chapter, you will perform the following steps:

1. Importing the ITSO Car Rental Application
2. Creating the Bundle Activator
3. Updating the MANIFEST file
4. Sending the OSGi Event from the Web Application
5. Handling the OSGi Event in the RCP application
6. Running and Testing the Updated Application

**Importing the ITSO Car Rental Application**
Follow the next steps to import the ITSO Car Rental application:

1. Download the ITSO Car Rental project interchange file from additional materials, see Appendix B, “Additional material” on page 721.
2. Start Rational Application Developer. **Start ➔ Run ➔ Programs ➔ IBM Software Development Platform ➔ IBM Rational Application Developer**
3. **Create** an empty workspace and choose **OK** to set the preferences to the Expeditor toolkit when prompted.
4. Select **File ➔ Import** from the menu.
5. From the Import dialog, select **Existing Projects into Workspace** and click **Next** to continue.

![Figure 13-2 Importing existing projects](image)

6. Choose the **Select archive file** radio button and click the **Browse** button to locate the project interchange file you downloaded earlier.
7. Click Select All and choose Finish to import all projects into your workspace.

Creating the Bundle Activator

A bundle activator class is a special class that is called by the OSGi framework during life-cycle events such as bundle start and stop. In this section, you will create a new bundle activator class for the Web application and implement its start() method where you will add code to return a reference of the OSGi EventAdmin service.

1. You will first need to create a bundle activator class for the Web application. Expand the com.ibm.itso.compapp.carrental.listcontracts project using the Package Explorer, right-click the com.ibm.itso.compapp.carrental.web package, and select New → Class from the context menu as illustrated in Figure 13-4.
2. In the **New Java Class** wizard, enter the following details and leave other entries to their default settings as illustrated in **Figure 13-5**.

   
   b. Name: `ListContractsActivator`
   
   c. Superclass: `org.eclipse.ui.plugin.AbstractUIPlugin`

3. Click Finish to generate and open the bundle activator class.
4. **Right-click** anywhere on the **ListContractsActivator** source editor and select **Source → Override/Implement Methods**.

5. From the Override/Implement Methods dialog, select the check box next to **start(BundleContext)** as shown in Figure 13-6.

![Override/Implement Method dialog](image)

**Figure 13-6 Override/Implement Method dialog**

6. Click **OK** to auto-generate the start method.

7. Add the following instance variables to the **ListContractsActivator** class:
   
   ```java
   private static BundleContext bc = null;
   private static EventAdmin ea = null;
   ```

8. You may see an error saying that EventAdmin cannot be resolved to a type. That is because you must add the import statement for that class. To resolve this error, right-click the class editor and select **Source → Organize Imports** and save the class.

9. In the **start(BundleContext arg0)** method, add the line `bc = arg0;` at the end as follows:
   
   ```java
   public void start(BundleContext arg0) throws Exception {
       super.start(arg0);
       bc = arg0;
   }
   ```

10. Now you need to add a method that returns the actual EventAdmin service. **Type** the following **getEventAdmin()** method in the **ListContractsActivator** class:
public static EventAdmin getEventAdmin()
{
    if (ea == null) {
        ServiceReference sr =
            bc.getServiceReference(
                EventAdmin.class.getName());
        ea = (EventAdmin)bc.getService(sr);
    }
    return ea;
}

11. Save and Close the ListContractsActivator class editor.

**Updating the MANIFEST file**

In order for the OSGi framework to use the bundle activator you just created, you need to define that in the bundle’s MANIFEST file.

1. **Open** the MANIFEST editor by double-clicking on the MANIFEST.MF file under the META-INF directory of the project com.ibm.itso.compapp.carrental.listcontracts.

2. In the MANIFEST editor, switch to the **Overview** tab (located at the bottom of the editor) and click the **Browse** button which is next to the Activator field.

3. In the dialog that appears, type **ListContractsActivator**, select the matching type from the list as shown in Figure 13-7, and click **OK** to continue.

![Figure 13-7  Selecting a type](image)
4. **Save** and **Close** the MANIFEST editor.

**Sending the OSGi Event from the Web Application**

You must now add code to the main Servlet class of the Web application in order to send an event when the user enters a customer number.

1. Also in the `com.ibm.itso.compapp.carrental.listcontracts` project, open the `ServContractList` class for editing by double-clicking on it under the `com.ibm.itso.compapp.carrental.web` package.

2. **Add** the following code at the end of the `doPost()` method:

   ```java
   Display display = PlatformUI.getWorkbench().getDisplay();
   display.asyncExec(new Runnable() {
       public void run() {
       }
   });
   ```

3. Press **Ctrl-Shift-O** to organize imports.

4. Inside the `run()` method you just added, **type** the following code which will create a new event and send the selected customer number:

   ```java
   Display display = PlatformUI.getWorkbench().getDisplay();
   display.asyncExec(new Runnable() {
       public void run() {
           Dictionary d = new Hashtable();
           d.put("CUSTOMER_NUMBER", custNum);
           Event eState = new Event("CUSTOMER_SELECTED",d);
           EventAdmin ea = ListContractsActivator.getEventAdmin();
           ea.sendEvent(eState);
       }
   });
   ```

5. Press **Ctrl-Shift-O** to organize imports and select the `org.osgi.service.event.Event` and `java.util.Hashtable` imports when prompted.
6. **Save** and **Close** the **ServContractList** class.

## Handling the OSGi Event in the RCP application

You now need to implement an OSGi event handler in the RCP application in order to listen to the event which is being sent by the Web application.

1. Expand the project `com.ibm.itso.compapp.carrental.booking` with the Package Explorer and open the **BookingComposite** class for editing.

2. Add the EventHandler interface by typing `implements EventHandler` to the class definition as follows:
   ```java
   public class BookingComposite extends Composite implements EventHandler {
   ```

3. Press **Ctrl-Shift-O** to organize imports and select the `org.osgi.service.event.EventHandler` import when prompted.

4. You should see an error in the **Problems** view that says “The type BookingComposite must implement the inherited abstract method EventHandler.handleEvent(Event)”. **Right-click** it and select **Quick Fix** as shown in Figure 13-9.
5. Select the **Add unimplemented method** option from the list and click **OK** to add the required handleEvent() method.

6. Find the new **handleEvent()** method at the bottom of the class and change its Event argument variable from **arg0** to **e**:

   ```java
   public void handleEvent(Event e) {
     // TODO Auto-generated method stub
   }
   ```

7. Inside of the **handleEvent()** method, **Replace** the line “// TODO Auto-generated method stub” with:

   ```java
   if (e.getTopic().equals("CUSTOMER_SELECTED")) {
       if(e.getProperty("CUSTOMER_NUMBER") != null) {
         String custNum = (String)e.getProperty("CUSTOMER_NUMBER");
         searchCustomer(custNum);
     }
   }
   ```
8. Registering the OSGi EventHandler by adding the following code to the constructor `BookingComposite(Composite, int, BookingView)` after the line `super(parent, style):

```java
Dictionary d = new Hashtable();
d.put(EventConstants.EVENT_TOPIC, new String[] {"*");
BookingActivator.getBundleContext().
   .registerService(EventHandler.class.getName(), this, d);
```

9. Press **Ctrl-Shift-O** to organize imports and select the `java.util.Hashtable` import when prompted.

![Organize Imports](image)

*Figure 13-11  Organize Imports*

10. **Save** and **Close** the `BookingComposite` class.

**Running and Testing the Updated Application**

Finally, you are going to create a launch configuration to run and test the updates that you made to the ITSO Car Rental application.

1. Select **Run → Run** from the RAD menu to open the Launch Configuration dialog.

2. Select **Client Services** on the left, **right-click** it and select **New** from the menu. Change the name of the configuration to **Car Rental OSGi**.
3. Click **Run** to launch the Lotus Expeditor client runtime.

4. Wait until the Lotus Expeditor client starts and select the ITSO Car Rental application from the application launcher.
5. Once the ITSO Car Rental application opens, you can test the new functionality you added in this chapter by entering one of the valid account numbers (1111, 1234, or 4321) in the Web application view, and after you click OK, you should see the RCP view refresh itself with the given account number as shown in Figure 13-14:

![Sample composite application](image)

Figure 13-14 Sample composite application

**Conclusion**

The OSGi Event Admin is a simple mechanism available to developers that create or migrate applications to Lotus Expeditor and who want to achieve intercommunication between different components such as Web and RCP views.
Managed composite applications

This chapter provides an introduction to managed composite applications and discusses the different ways to lay out a composite application. It also covers how to provision components to your desktop clients as well as how to implement access control and roles.

In this chapter you will find information about the following topics:

- **Overview**: Briefly sketches the parts and pieces of a managed application.
- **Prerequisites for declaring composite applications**: Lists the mandatory products and technologies.
- **Access control and roles**: Shows how to restrict certain capabilities for a group of people, while allowing them for another group.
- **Provisioning**: Explains how to provide the downloadable code.
- **Definition and placement**: Describes the principles of assembling components together in a page.
- **Providing components in a portal sense**: Shows how to perform this using the available portal tools.
- **Laying out applications programmatically**: Provides a guide to perform this for non-connected clients.
14.1 Overview

Using WebSphere Portal tools, an application becomes a managed application because it is centrally administered from the server side, and deployed to Lotus Expeditor clients automatically and incorporates the roles of a user. WebSphere Portal and Lotus Expeditor share relationships at different levels. Figure 14-1 shows the main building blocks that collaborate in this environment.

![Figure 14-1 Architectural view of Expeditor and WebSphere Portal]

Before we dive into the details of composing an application layout, you will need to understand the concepts of managed applications, how they are installed on the server, how they get distributed and what pieces are required.

Figure 14-2 depicts an architectural view of a managed application.
Figure 14-2 illustrates the various pieces of a managed composite application. The main parts are:

- Access control settings for the application, pages and components
- Provision of the components
- Definition and placement of the components
- Optional parameters of the components
- Optional declaration of wires between components

### 14.2 Prerequisites for declaring composite applications

In order to be able to declare composite applications with WebSphere Portal, WebSphere Portal V6.0 in addition to some other components, must be installed on your system. The complete steps to install all required products go beyond the scope of this book, but here we present the simplified steps to satisfy these requirements:

- Install WebSphere Portal V6.0.
- Install a HTTP server that is supported by WebSphere Portal 6.0.
- Install the WebSphere Application Server plug-in for your HTTP server.
- Configure the plug-in, in WebSphere Application Server.
- Install Lotus Expeditor Network Client Installation (NCI).

This provides you with the necessary tooling in WebSphere Portal and provides the required files and components on your HTTP server to allow users to install the Lotus Expeditor client run time by accessing the WebSphere Portal.
14.3 Access control and roles

An important feature of a managed application is its capability to provide access control not only to the application itself, but also to parts of the application such as pages and components that form the complete application. Figure 14-2 on page 349 shows a simple application consisting of four components all being placed on one page. But what if the component on the lower right is supposed to be seen only by a particular group of people? To accomplish this, access controls can be used to prevent people from getting parts of applications that are not intended for them.

When managing applications with WebSphere Portal, the security features of WebSphere Portal are used to accomplish access control to applications. With WebSphere Portal v6.0, composite applications are introduced to WebSphere Portal. With it came the concept of roles.

While using the composite application editor inside WebSphere Portal for each application, you can create roles that allow you to specify who has access to which component in your application. If users have no access to any of your components, they will not see the application listed in the application catalog.

If users have access to at least one component, they will be able to see the application in the catalog and thus install the application on Lotus Expeditor. However, only the features required by the component for these users will be
installed on the Lotus Expeditor client. Figure 14-3 shows an example of a role definition for a sample application.

Figure 14-3   Sample definition of roles for an application

In Figure 14-3, users for the role Users will only have access to the first four components of this application.

To ease the assignment of access control to applications, it is recommended that you create groups in your directory representing the roles used in your applications. This way, you do not have to edit the application (or its template) to either allow or withdraw users for your application.

Note: Only the group in your directory has to be administered, and it can be done by someone completely uninvolved with the composite application design.
14.4 Provisioning

Besides letting Lotus Expeditor know what components should appear on a certain page, a managed application also means that installation of the application takes place automatically when the user accesses the application for the first time. Additionally, if components used in the applications are not components that come with the base installation of Lotus Expeditor, then they must be retrieved by Lotus Expeditor in order to be available for local execution.

The management of such components is performed by the Open Service Gateway initiative (OSGi) layer of the underlying Eclipse platform. Lotus Expeditor only retrieves the information about features that has to be made available and from which location these features have to be retrieved. This information is then passed to the OSGi layer, which in turn retrieves the installation files, and installs the features and their corresponding plug-ins in order to start them appropriately.

The place to store features and plug-ins in order to provision them to an Eclipse based client is called an update site. Since Eclipse v3.0, update sites are used to install new features into Eclipse and to update existing ones. An update site is usually a very simple file based repository. Update sites can be accessed through either local file access or via the HTTP protocol. Contents of an update site are managed in an XML based file, called site.xml, residing at the root of an update site location.

Example 14-1 shows an example of an update site tree.

Example 14-1  Directory listing of an update site

```
|    site.xml
|   +--features/
|       |   com.ibm.itso.compapp.carrental.booking.feature_1.0.0.jar
|       |   com.ibm.itso.compapp.carrental.choosecar.feature_1.0.0.jar
|       |   com.ibm.itso.compapp.carrental.creditcard.feature_1.0.0.jar
|       |   com.ibm.itso.compapp.carrental.listcontracts.feature_1.0.0.jar
|   +--plugins/
|       |   com.ibm.itso.compapp.carrental.booking_1.0.0.jar
|       |   com.ibm.itso.compapp.carrental.choosecar_1.0.0.jar
|       |   com.ibm.itso.compapp.carrental.creditcard_1.0.0.jar
|       |   com.ibm.itso.compapp.carrental.database_1.0.0.jar
|       |   com.ibm.itso.compapp.carrental.listcontracts_1.0.0.jar
```

Important: Notice that in this example there are five plug-ins but only four features. Features can contain multiple plug-ins. In this example, the booking feature contains the booking and the database plug-ins.
The site.xml file for this update site, shown in Example 14-1, is illustrated in Example 14-2.

Example 14-2   Contents of a site.xml file describing an update site

```xml
<?xml version="1.0" encoding="UTF-8"?>
<site>
  <feature
    url="features/com.ibm.itso.compapp.carrental.booking.feature_1.0.0.jar"
    id="com.ibm.itso.compapp.carrental.booking.feature" version="1.0.0"/>
  <feature
    url="features/com.ibm.itso.compapp.carrental.choosecar.feature_1.0.0.jar"
    id="com.ibm.itso.compapp.carrental.choosecar.feature" version="1.0.0"/>
  <feature
    url="features/com.ibm.itso.compapp.carrental.creditcard.feature_1.0.0.jar"
    id="com.ibm.itso.compapp.carrental.creditcard.feature" version="1.0.0"/>
  <feature
    url="features/com.ibm.itso.compapp.carrental.listcontracts.feature_1.0.0.jar"
    id="com.ibm.itso.compapp.carrental.listcontracts.feature" version="1.0.0"/>
</site>
```

When an update site location is made available to an Eclipse based product, such as Lotus Expeditor, it reads the contents of the site.xml file and allows you to install or update features from this location.

Note: As Lotus Expeditor is a managed client, users are not given the freedom to choose update sites of their own; however, application deployers can specify from what location the features required to run their applications can be downloaded. An example on how to configure the update site location is shown in Figure 14-7 on page 359.

14.5 Definition and placement

The way to compose an application with WebSphere Portal is to place portlets on a page. This is similar to creating composite applications that are supposed to run on WebSphere Portal itself, and therefore are accessed as a client to portal using a browser. Lotus Expeditor is also able to run more component types than WebSphere Portal and this is because WebSphere Portal deals with portlets only.

Therefore, when running composite applications in Lotus Expeditor, you require a way to specify what component is to be run when there is a portlet component on a certain page.
The component types, which can be used for creating composite applications to run in the Lotus Expeditor platform, have been described in detail in Chapter 4, “Components in composite applications” on page 69. To summarize, these are:

- Eclipse based Rich Client Platform (RCP) components, regardless of whether they are Standard Widget Toolkit (SWT) or SWT/JFace components.
- Portlets written to the JSR168 specification can be executed in Lotus Expeditor’s local portlet container.
- Web Services Remote Portlets (WSRP) executed on a remote portal server but displayed through Lotus Expeditor’s local portlet viewer.
- Web applications such as servlet and JavaServer Pages (JSP) based applications running on Lotus Expeditor’s local Web container and displayed by the embedded browser control.

The following sections describe the requirements for placing components on a page, and how to do this for the various types of components.

### 14.5.1 Portlet instances versus portlet clones

In this section, we describe how portlets can be used as placeholders to represent other component types when used in conjunction with Lotus Expeditor. In some cases, there is no need to have one portlet per component. Also, unless you do not want to provide component specific parameters for a particular portlet, you can use multiple instances of the same portlet.

The *Rich Client Layout Administration Portlet* takes care of injecting the mandatory component specific settings when Lotus Expeditor requests the page layout. However, if you plan to provide custom settings with your component, you must provide these settings as portlet parameters. You do this by using the WebSphere Portal portlet administration page. Portlet parameters can only be specified at the portlet level, and you are required to create a clone (for example, the *Rich Client View Placeholder* portlet), and name it accordingly to your needs and set your custom parameters as required.

Custom parameters are always a hard requirement for cloning portlets. Additionally, there is a soft requirement due to the fact that portlet instances go by the same name on the page. In this case, it is not so easy to determine which portlet represents which component, especially when working with the role editor tool where you only see the Rich Client View Placeholder listed multiple times and it does not tell you what component it is referring to. Therefore, it is a good practice to always clone the portlet and rename it accordingly to the component it represents.
14.6 Providing components in a portal sense

This section describes how components are placed on a page when using the portal administration tool.

14.6.1 Placing Eclipse based components

In general, Eclipse based components are plug-ins in Eclipse's terminology or bundles in OSGi's terminology. However, you must be aware that Eclipse based components are an alien construct to WebSphere Portal in the sense that they cannot be installed in WebSphere Portal and they cannot be executed by WebSphere Portal.

Therefore, in order to work with these components in a way that WebSphere Portal understands, a proxy portlet is used to stand in for the real component. This support becomes available after installing the Lotus Expeditor NCI and thus the Rich Client View Placeholder portlet becomes available to WebSphere Portal. This portlet can be placed on a page to act as a placeholder for an Eclipse view.

You use the Rich Client Layout Administration portlet to specify settings such as what view should be used, what features are contained in the plug-ins for this

---

Note: Cloning and renaming portlets is also a good practice when working with the **WSRP Rich Client Viewer Enablement Portlet** and the **Rich Client Managed Browser Administration portlet**. Both portlets are described later in this chapter.
view, how big it should be, and so on. Figure 14-4 shows an example of a layout page containing two components.

![WebSphere Portal page layout editor](image)

**Figure 14-4  WebSphere Portal page layout editor**

Both portlets shown in Figure 14-4 are instances of the Rich Client View Placeholder portlet. To let Lotus Expeditor know what view it should display and what the details are, you must switch to the rich client tab of the layout page that holds the Rich Client Layout Administration portlet (see Figure 14-5).

The rich client tab allows you to specify global options for the page and component related options for each of the components you have placed on the page. The global options are:

- **Visible** indicates that the page can potentially be shown.
- **Add to launcher** indicates that this page will be shown in the application selection switch.
- **Open automatically when the application starts** indicates that the page will open with no action required from the user.
- Hidden indicates that the page will not be shown in the application launcher or in the navigator section.
- Icon for application switcher indicates the icon (JPG, GIF or PNG) that should graphically represent this page in the application selection switch.
- Activities is a comma-separated list of activities that are to be included when the perspective is opened on the client system.
- Default folder is the name of the folder where views should be placed in, if not specified at the view level.

![Rich Client Layout Administration](image)

**Figure 14-5 Rich Client Layout Administration**

At the bottom of the global options in Figure 14-5, you find a list of the components that you can add to the page. The list allows you to change the portlets or the portlet instance’s parameters or both. These parameters will be
injected into the page, when Lotus Expeditor requests the page to retrieve the settings for this composite application page. Figure 14-6 illustrates the definition of the booking portlet in the car rental sample application.

Figure 14-6   Rich client settings for an RCP component
In Figure 14-6, the instance description represents the title for the Eclipse view window. Depending on the portlet type that this instance represents, you get different options for the rich client properties. In this example, it is representing an RCP view and therefore the option This portlet represents an SWT view on the rich client is selected and the Eclipse view ID is also provided.

Note: Configuration options for portlet components are explained in 14.6.2, “Placing portlets as components” on page 360.

Lotus Expeditor must be able to retrieve all the required features and plug-ins to function properly. In the general case, you can bundle all the required plug-ins into a single feature. However, you can also create multiple features when having everything in one place does not serve your purpose.

The available options allow you to specify the feature name and the desirable feature version. You can also configure what matching rule Lotus Expeditor should use to verify the available version to be selected. Most importantly, the update site provisioning URL, from where this feature is downloaded, must be entered. A sample configuration window is shown in Figure 14-7.

![Rich Client Layout Administration](image-url)
The available matching rules are illustrated in Table 14-1.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>perfect</td>
<td>Must be exactly to what you specified</td>
</tr>
<tr>
<td></td>
<td>Example: 1.0.0</td>
</tr>
<tr>
<td>equivalent</td>
<td>Can be at a higher service level</td>
</tr>
<tr>
<td></td>
<td>Example: 1.0.1, 1.0.2, 1.0.9, 1.0.35</td>
</tr>
<tr>
<td>compatible</td>
<td>Can be at a higher minor version level</td>
</tr>
<tr>
<td></td>
<td>Example: 1.2.0, 1.3.4, 1.12.0</td>
</tr>
<tr>
<td>greaterOrEqual</td>
<td>Can be anything that is equal or higher to the specified level</td>
</tr>
<tr>
<td></td>
<td>Example: 1.0.0, 1.0.1, 2.1.0, 3.2.8</td>
</tr>
</tbody>
</table>

14.6.2 Placing portlets as components

When using the page layout editor in the portal administration tool to place portlets on a page, the portlet needs to be made available to WebSphere Portal. In order to do this, you must export the portlet as a WAR file and install it into WebSphere Portal. Once you have done this, you are required to place the portlet on the page in a similar way that you want a portlet to be used by WebSphere Portal.

In the Rich Client Layout Administration portlet, you specify how Lotus Expeditor obtains the feature where this portlet is bundled in. In addition, you specify the context root, as shown in Figure 14-8.

As shown in Example 14-3, the context root is located in the plugin.xml file.

Example 14-3 plugin.xml excerpt from a Client Services Portlet project

```
<extension point="com.ibm.pvc.webcontainer.application">
  <contextRoot>
```
14.6.3 Placing a WSRP portlet viewer

When using the WSRP portlet viewer as a component on your page, you have to make use of a proxy portlet provided by the Lotus Expeditor’s NCI tool. The portlet is called *WSRP Rich Client Viewer Enablement Portlet*. This portlet can be used in multiple places and each place gets its own instance properties.

As shown in Figure 14-9, when you initially place the portlet on a page, the unconfigured state is displayed.

![Figure 14-9  WSRP Rich Client Viewer enablement portlet](image)

As shown in Figure 14-10, activating the portlet’s menu and selecting the Configure option will bring you to the first option window.

![Figure 14-10  WSDL address and security options](image)
Fill in the URL for your WebSphere Portal's fully-qualified domain name followed by /wps/wsdl/. The remainder of the URL is at your disposal, and you can choose any arbitrary name. Once you select OK, the window shown in Figure 14-11 allows you to select the remote portlet that you want to show.

![Figure 14-11 Remote portlet selection](image)

The window shown in Figure 14-12 provides the option to specify whether a clone of the portlet is required to display the portlet.

![Figure 14-12 Selecting if a clone is needed](image)

Select OK to conclude the process to add a WSRP portlet on a composite application page.

### 14.6.4 Placing a Web application as a component

Since they do not require any extra coding, the process to add a Web application to a page is a simple task. Once your Web application project is available and you have packaged it into a feature, the remaining tasks are:

- Specify where to place a browser instance
- Enter the URL of the Web application

As with RCP components, a placeholder is needed to indicate the place on the portal page layout. The Lotus Expeditor's NCI tool installs a portlet called *Rich*
Client Managed Browser Administration. This portlet can either be duplicated or used as multiple instances. Once the portlet is placed on the page, the available options are also be displayed.

Most of the options speak for themselves. The majority of them allows you to change the appearance of the browser, the controls you want the user to have available and the security settings for running executable code within the browser.

The two most relevant settings for Web application components are:

- Initial URL. This value is the context root that has been assigned to your Web application. As shown in Example 14-4, the context root is found in the project’s plugin.xml file.

Example 14-4 plugin.xml excerpt of a Client Services Web Project

```xml
<plugin>
  <extension point="com.ibm.pvc.webcontainer.application">
    <contextRoot>
      /com.ibm.itso.compapp.carrental.listcontracts
    </contextRoot>
    <contentLocation>WebContent</contentLocation>
  </extension>
</plugin>
```

- URL points to local Web application. This option must be set to Yes so that the embedded browser retrieves the port settings of your current Lotus Expeditor instance and injects it into the URL appropriately.

Note: The embedded Web container starts listening on a random port every time you restart Lotus Expeditor client. Therefore, it is necessary to free up the URLs from a port number. If you have good reasons to have your Lotus Expeditor use a particular port number every time it starts, refer to Configuring the Web Container properties in the Assembling and Deploying Lotus Expeditor Applications Guide.

14.7 Specifying custom settings

Composite application components often implement a construct that uses the same piece of code to show different content depending on an additional parameter given to the component at initialization time.
A common scenario is, for example, components that show two different Web pages on the same page. In this scenario, there is no need to write two different components. The embedded browser can handle this by showing the proper Web page and you only are required to provide an additional setting for the component indicating what Web page should be loaded.

For the embedded browser, a portlet is provided to allow you to place embedded browser components that can be highly customized.

To provide settings to your plug-ins that can be used during run time, you have to add settings as parameters to your portlet. Figure 14-13 shows the portlet parameter window.

![Figure 14-13 Example portlet parameters](image-url)
If required, the parameters specified in Figure 14-13 can be programmatically obtained in your plug-in code via a construct. Example 14-5 illustrates how you can do this.

Example 14-5  Obtaining portlet parameters inside a plug-in

```java
TopologyHandler handler;

BundleContext context = BookingActivator.getBundleContext();
ServiceReference ref = context.getServiceReference(TopologyHandler.class.getName());

if (ref != null)
    handler = (TopologyHandler) context.getService(ref);

if (handler != null) {
    String componentDataIdentifier = this.getViewSite().getSecondaryId();
    ComponentData data = handler.getComponentData(componentDataIdentifier);
    if (data != null) {
        String[] strdata = data
                        .getPreference("com.ibm.itso.compapp.carrental.booking.manager");
        if (strdata != null && strdata.length > 0)
            if (strdata[0] != null)
                manager = (strdata[0].compareToIgnoreCase("false") == 0 ||
                           strdata[0].compareToIgnoreCase("no") == 0)
                    ? false : true;
    }
}
```

14.8 Templates

When multiple composite applications are similar in a layout and components need to be assembled, creating layouts using WebSphere Portal leads to a considerable amount of effort. Also, in many cases, these composite applications will normally differ in a few minor attributes.

In this scenario, a good practice is to restrict yourself to create the least common denominator of all applications and take it as a basis to create more complete applications. This process of this type of application design is called **templating**.

Application layouts created on WebSphere Portal with the mechanisms described in this chapter can be exported as a template. These templates can be used as a base to create new composite applications and these applications will inherit all the work that has been previously done, without requiring the recreation of the steps again.
14.8.1 Creating a template

To create a template for your composite applications, open your application in the Template Editor, and select the menu on the left most section of your application page. Figure 14-14 illustrates the option to save your application as a template.

![Figure 14-14  Saving an application as a template](image)

Select **Save Application as Template** to obtain the window shown in Figure 14-15 and enter the application template name and optional description.

![Figure 14-15  Specifying the template name](image)

After you have completed this step, your template will be available to create new applications. Go to the WebSphere Portal template administration page, and
select **New** to create a new application. The template that you have previously generated will now be available as a basis for your future applications.

Figure 14-16 illustrates the window to create a new application using a template.

![Figure 14-16 Creating an application based on a template](image)

### 14.9 Laying out applications programmatically

Considering that one of the real benefits of a composite application is the reuse of components, laying out applications programmatically is of no real benefit. However, there are situations where you cannot afford a declarative approach, for example, because of a size, connectivity or any other deployment related reasons.

Using a perspective to layout your application is the easiest way to produce an immediate result. Additionally, for testing and debugging purposes, it might be appropriate to use this way of assembling your application. Perspectives are part of the basic Eclipse framework that Lotus Expeditor is built on.

**Note:** You can find more detailed information about how to work with perspectives in the online help of Eclipse Plugin-Development Help and in various publications on the eclipse.org Web site.

In the sample application described in this book, a plug-in on its own (com.ibm.itso.compapp.;carrental.application) is used to provide the composition
of the components. Generally, three things are necessary to completely define an application with its perspective:

- The perspective class, which is the Java implementation linked to the org.eclipse.ui.perspective extension point.
- The definition of the extension in the plug-in’s plugin.xml file.
- A reference to the definition of the perspective in the declaration of a Lotus Expeditor application.

14.9.1 Implementing a perspective

Implementing a perspective means creating a class that implements the IPerspectiveFactory interface defined in org.eclipse.ui. The mandatory function createInitialLayout is used to programmatically layout the various pieces on the composite application. The region consists of various areas where the editor area is the one you are dealing with. You add views to this area by invoking the function.addView, which takes as parameters the view ID you want to add, the position relative to an existing item, the ratio you want this existing item to allocate and the name or ID of the already existing item.
The sample car rental application layout is shown in Figure 14-17.

![Sample composite application](image)

**Figure 14-17  Sample composite application**

The `createInitialLayout` function for the sample application is illustrated in Example 14-6.

**Example 14-6  The `createInitialLayout` function of an `IPerspectiveFactory`**

```java
public void createInitialLayout(IPageLayout layout) {
    String editorArea = layout.getEditorArea();
                  IPageLayout.TOP, IPageLayout.RATIO_MAX, editorArea);
                  IPageLayout.RIGHT, 0.60f,
```

Chapter 14. Managed composite applications  369
The following is a description of the process to implement the `createInitialLayout` method:

- Add the view `com.ibm.itso.compapp.carrental.booking.BookingView` onto the editor area.
- Add the view `com.ibm.rcp.portletviewer.portletview:com.ibm.itso.compapp.weather.wsrp` to it, allowing the `BookingView` 60% of the horizontal space. Since this is a view displaying a portlet, you must specify what portlet to display and you do this by specifying a secondary view ID that you separate from the primary view ID with a colon.
- The third component is located below the `BookingView` and its ID is `com.ibm.compapp.carrental.creditcard.CreditCardView`. In this sample scenario, you provide the `BookingView` 65% of the vertical space.
- Lastly, you add a second portlet viewer below the previous portlet component, allowing both to take up 50% of the vertical space. Again, you specify a secondary ID to let the portlet viewer know which portlet you want it to display.

Where do secondary IDs come from?

When you create a portlet component, you provide such information as the portlet name and the context root. In order to provide this information through a specific ID, you make use of another Eclipse extension point called `com.ibm.rcp.portletviewer.portlets`. This extension point holds various tags that lets the portlet viewer know exactly what to display. The declaration of the sample car selection portlet is shown in Example 14-7.

```
Example 14-7  Sample car selection portlet component extension point

...<extension point="com.ibm.rcp.portletviewer.portlets"><portletData
```
The context root is defined in the portlet plugin.xml file. You find it in the extension point com.ibm.pvc.webcontainer.application. The tag name is contextRoot. Example 14-8 shows the extension point.

Example 14-8  plugin.xml excerpt of a Client Services Portlet project

```xml
<extension point="com.ibm.pvc.webcontainer.application">
  <contextRoot>
    /./com.ibm.itso.compapp.carrental.choosecar
  </contextRoot>
</extension>
```

The portlet name is defined in the portlet options tag, the tag name is portlet-name. For example:

```xml
<portlet-name>PortSelectCar</portlet-name>
```

The value of portletwindowid must always be default.

### 14.9.2 A sample remote portlet component

For the sample weather portlet component included in the sample application illustrated in this book, a different approach was taken to illustrate an example of a portlet being displayed locally, but actually running on a remote portal server. This is accomplished by using the WSRP viewer portlet.

**Note:** WSRP is a standard defined by the OASIS (Organization for the Advancement of Structured Information Standards) consortium.

In this scenario, it is assumed that a WSRP compliant portlet already exists on the portal server. Since no programmatic code is required, all you need to do is create a copy of the WSRP Rich Client Viewer Enablement Portlet and place it on a composite application page. Once you have done this, you navigate through this page, bring in the portlet into configuration mode and specify the parameters.
to be used by accessing this proxy portlet. The configuration options are listed in Table 14-2.

Table 14-2  Remote portlet configuration options

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>WSDL address</td>
<td>http://&lt;your-FQDN&gt;:&lt;port&gt;/wps/wsdl/&lt;your-own-unique-name&gt;.wsdl</td>
</tr>
<tr>
<td></td>
<td>For example: <a href="http://lo-s630.ibm.com/wps/wsdl/itosweather-portlet.wsd1">http://lo-s630.ibm.com/wps/wsdl/itosweather-portlet.wsd1</a></td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> In this example a port is not specified since the standard port 80 is used.</td>
</tr>
<tr>
<td>Security Options</td>
<td>No Security</td>
</tr>
<tr>
<td></td>
<td>LTPA</td>
</tr>
<tr>
<td></td>
<td>Provide WSRP by a third-party producer (using the Username token)</td>
</tr>
<tr>
<td>Portlet</td>
<td>Select one of the listed portlets</td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Only portlets that have been enabled as WSRP portlets will be listed here.</td>
</tr>
<tr>
<td>Clone</td>
<td>A portlet clone is needed.</td>
</tr>
</tbody>
</table>

For the specific case of a programmatic application layout, the page does not have to be accessible by the client. When you copy the portlet, the portlet gives itself a new parameter called a **handle**. This parameter can be obtained from the WebSphere Portal server administration interface that manages the portlet parameters.

All the extension point is required to know is the portlet WSDL file location, the portlet handle value, whether cloning the portlet is needed and what kind of security mechanism, if any, you must implement to authenticate to the portal server. All this information is obtained from WebSphere Portal server. Example 14-9 illustrates a sample configuration,

*Example 14-9  plugin.xml excerpt describing the parameters for a WSRP viewer*

```xml
...<extension_point="com.ibm.rcp.portletviewer.WsrpPortlets">
  <wsrpData entityID="com.ibm.itso.compapp.weather.wsrp"
    wsrp_wsd1="http://lo-s630.ibm.com/wps/wsdl/itsowsrpweather.wsd1"
    handle="5_8GUNUKG100US502E64VV9H2007"
    need_client_clone="true"
    isSecured="false" />
</wsrpData>
```
14.9.3 Declaring the perspective

To declare the perspective, you have to add a section in the plugin.xml file declaring an extension to the org.eclipse.ui.perspectives extension point. This is performed directly and it only requires you to provide an ID that you later use to identify this perspective. You can choose any arbitrary name here; however, it is a good practice to use the plug-in name as a prefix and append the perspective name to it. A sample declaration is shown in Example 14-10.

Example 14-10  plugin.xml excerpt describing an Eclipse perspective

```xml
<extension point="org.eclipse.ui.perspectives">
  <perspective
class="com.ibm.itso.compapp.carrental.application.CarRentalPerspective"
  id="com.ibm.itso.compapp.carrental.application.CarRentalPerspective"
  name="ITSO Car Rentals" />
</extension>
```

14.9.4 Declaring the application

The plug-in definition file contains an extension with the extension point com.ibm.eswe.workbench.WctApplication. This extension point indicates that the plug-in is a Lotus Expeditor application. This definition requires the ID of a perspective and a display name. Optionally, an icon image can also be specified to graphically represent the application in the Lotus Expeditor application switcher menu. Example 14-11 shows an example of the extension point declaration.

Example 14-11  plugin.xml excerpt of a Client Services Web project

```xml
<extension id="main_view" point="com.ibm.eswe.workbench.WctApplication">
  <Application
    DisplayName="%application.name"
    Icon="icons/ITSO.png"
    PerspectiveId="com.ibm.itso.compapp.carrental.application.CarRentalPerspective" />
</extension>
```
**Note:** The `displayName` tag starts with a `%` sign. It indicates that the representation of this tag will be taken from the `plugin.properties` file allowing internationalization. Multiple `plugin.properties` files can be named accordingly to the language used, for example `plugin_de_DE.properties` for German and `plugin_pt_BR.properties` for Brazilian Portuguese.
Exporting components and publishing the update site

This chapter provides step-by-step instructions to export components as WAR files and publish the update site. Component WAR files are used by tools to assemble composite applications. The update site is used for composite application provisioning.

This chapter provides the following topics:

- Importing components into Rational Application Developer
- Creating feature and update site projects
- Exporting components as WAR files
- Publishing an update site
15.1 Overview

When using the WebSphere Portal Tools with the Network Client Installer to assemble composite applications, you will need to import components as WAR files. In addition, if you are planning to deploy composite applications from an Update Site server you will need to create an update site project using Rational Application Developer and publish the project in an HTTP server.

Figure 15-1 illustrates what needs to be done before you assemble managed applications using the tools in WebSphere Portal and what is needed to publish the update site for provisioning of composite applications.

![Diagram illustrating the process of exporting components and the update site project](image)

Figure 15-1: Exporting components and the update site project

15.2 Steps for provisioning composite applications

Follow the steps in the next five tasks to create an update site and deploy composite applications:

- “Importing the composite application components” on page 377
- “Creating the component features” on page 378
- “Composite application update site” on page 384
- “Creating the stub portlet for the SWT component” on page 387
- “Exporting the deployment artifacts” on page 393
15.2.1 Importing the composite application components

In Rational Application Developer, if the components are not available in your workspace, follow these steps to import the application components:

1. Download the DemoAppRCP.zip file available from additional materials to a local directory,
2. Start your development environment. In this sample scenario, Rational Application Developer and the Lotus Expeditor Toolkit.
3. Select **File → Import** from the menu bar.
4. Select **General → Existing Projects into Workspace** and click Next.
5. Choose **Select archive file** and enter the location for the DemoAppRCP.zip file. Select the SendMessage, ReceiveMessage and SWTMessage components, as shown in Figure 15-2.

![Figure 15-2 Importing component projects](image)

6. Click Finish to import the components.
7. Once the components are imported into your workspace, open the build.properties file for the SWT Message component and make sure that the SWTMessage.wsdl file is included in the binary build, as illustrated in Figure 15-3. If not included, select it and save the changes.
15.2.2 Creating the component features

For proper deployment, components must be available from an update site. In addition, each component has to be packaged as a feature and these features are part of the update site.

**SendMessage feature**

Follow these steps to create the SendMessage component feature:

1. Select File -> New -> Other from the menu bar.
2. Select Plug-in Development -> Feature Project and click Next.
3. Enter the details shown in Figure 15-4 and click Next.
4. Select the `SendMessage` component from the plug-ins list as shown in Figure 15-5. Click Finish to create the `SendMessage` feature.
5. If the Open Associated Perspective prompt appears, click Yes to switch to the Plug-in development perspective.

6. The SendMessage feature is created, as shown in Figure 15-6. You can close the feature.xml editor.

---

**ReceiveMessage feature**

Follow these steps to create the ReceiveMessage component feature:

1. Select File -> New -> Other from the menu bar.
2. Select Plug-in Development -> Feature Project and click Next.
3. Enter the details shown in Figure 15-7 and click Next.

![New Feature dialog box](image)

**Figure 15-7   ReceiveMessage component feature**

4. Select the ReceiveMessage component from the plug-ins list as shown in XXX. Click Finish to create the ReceiveMessage feature.
5. The ReceiveMessage feature is created, as shown in Figure 15-9. You can close the feature.xml editor.

**Figure 15-8   ReceiveMessage plugin**

**Figure 15-9   ReceiveMessage feature general information**

**SWTMessage feature**

Follow these steps to create the SWTMessage component feature:

1. Select *File -> New -> Other* from the menu bar.
2. Select *Plug-in Development -> Feature Project* and click *Next*.
3. Enter the details shown in XXX and click *Next*. 
4. Select the SWTMessage component from the plug-ins list as shown in Figure 15-11. Click Finish to create the SWTMessage feature.
5. The SWTMessage feature is created, as shown in XXX. You can close the feature.xml editor.

![General Information](image)

*Figure 15-12  SWTMessage feature general information*

### 15.2.3 Composite application update site

The next step is to create the update site for the sample composite application. Follow these steps to create the update site:

1. Select **File -> New -> Other** from the menu bar.
2. Select **Plug-in Development -> Update Site Project** and click Next.
3. Enter the details shown in Figure 15-13 and click Finish.
Chapter 15. Exporting components and publishing the update site

4. The update site is created and the site.xml descriptor is opened as shown in Figure 15-14.

5. In the site.xml descriptor, click **Add Feature**. Enter com.ibm.itso as a filter and select the SendMessage, ReceiveMessage and SWTMessage features as shown in Figure 15-15. Click **OK**.
6. The features will be added to the update site descriptor, as shown in Figure 15-16. Click **Build All** to build the update site contents.

![Feature Selection](image1)

**Figure 15-15  Feature selection**

7. Wait until the features are compiled and the update site is constructed. When the update site is created, it will include the selected features as shown in Figure 15-17.

![Update Site Map](image2)

**Figure 15-16  Update site project added features**
15.2.4 Creating the stub portlet for the SWT component

When you assemble a composite application using the portal tools, an SWT component needs to be represented by a proxy portlet as follows:

- If the SWT component does not participate in component intercommunication using wires, the Rich Client RCP View Placeholder portlet, installed as part of the Network Client Installer, can be used to represent the SWT component.

- If the SWT component intercommunicates with other portlet components using wires, you will need to create a stub portlet to be used on behalf of the SWT component. This stub portlet must include the following:
  - The WSDL file with input and output property definitions of the SWT component.
  - The location of the WSDL file in the portlet descriptor portlet.xml

Note: The procedure described in this section requires Rational Application Developer and the Expeditor Toolkit as a development environment.

Follow these steps to create the stub portlet for the SWTMessage component:

1. Select File -> New -> Project from the menu bar.
2. Check Show All Wizard, select Portal -> Portlet Project and click Next.
3. If the Confirm Enablement window appears, click OK to activate the portal development support.
4. Uncheck the **Add to project to an EAR** option and enter the details shown in Figure 15-18. This portlet will never be executed but Portal Tools will read the WSDL file to create wires. Therefore, any target runtime and portlet type can be used. Click Next.

![New Portlet Project](image)

*Figure 15-18  Creating the SWT stub portlet project*

5. By default, the generated portlet supports view mode only as shown in Figure 15-19 and is all that you need for this scenario.

6. Uncheck the Generate a custom portlet class option and **click Finish** to create the portlet.
7. If the Open Associated Perspective window appears, click No.
8. The view mode JSP file will open in the JSP editor. Change the default Place content here message for SWT Message Proxy portlet as shown in Figure 15-20. Save the changes.
9. Expand the SWTMessage stub portlet project. Right-click the WebContent folder and select **New -> Folder**. Enter `wsdl` as the folder name and **click Finish** to create the `wsdl` folder.

10. Copy the `SWTMessage.wsdl` file from the SWTMessage component to the `wsdl` folder in the SWTMessage stub portlet, as shown in Figure 15-21.

11. Open the SWTMessage stub portlet `portlet.xml` file to update it with the WSDL file location and required preferences.
12. Click the com.ibm.itso.casample.swtmessage.portlet portlet and scroll down to the Persistent Preferences Store section, shown in Figure 15-22. Click Add.

![Figure 15-22 Persistent preferences](image)

13. Enter `com.ibm.portal.propertybroker.wsdllocation` as the preference name and the WSDL file location as the preference value, as shown in Figure 15-23. Click OK.
14. Click **Add** again to add another preference. Enter `com.ibm.rcp.viewId` as the preference name and `com.ibm.itso.demoapp.swtmessage.MainView` as the preference value, as shown in Figure 15-24. Click **OK**.

15. Verify that both preferences are listed as preferences in the portlet descriptor (portlet.xml) as shown in Example 15-1. Save the changes.

**Example 15-1  Portlet preferences in portlet descriptor**

```xml
<portlet-preferences>
  <preference>
    <name>com.ibm.portal.propertybroker.wsdllocation</name>
    <value>/wsdl/SWTMessage.wsdl</value>
  </preference>
  <preference>
    <name>com.ibm.rcp.viewId</name>
    <value>com.ibm.itso.demoapp.swtmessage.MainView</value>
  </preference>
</portlet-preferences>
```
15.2.5 Exporting the deployment artifacts

The Portal tools used to assemble composite applications need to have the components available. Therefore, you need to export components as WAR files from your development environment. In this sample scenario, the following artifacts need to be exported for deployment:

- SendMessage portlet WAR file
- ReceiveMessage portlet WAR file
- SWTMessage stub portlet WAR file
- Update Site for the components

**Export the SendMessage portlet**

Portlets are exported as WAR files. Follow these steps to export the SendMessage portlet:

1. Right-click the com.ibm.itso.demoapp.sendmessage.portlet project and select **Export -> Web -> WAR file**.

2. Create a folder named CASAMPLE as destination for the WAR file, as shown in Figure 15-25. Click Finish to export the WAR file.

![Export portlet as a WAR file](image)
Export the ReceiveMessage portlet
In a similar way, export the ReceiveMessage portlet:

1. Right-click the com.ibm.itso.demoapp.receivemessage.portlet project and select Export -> Web -> WAR file.
2. Select the CASAMPLE folder as the destination for the WAR file. Click Finish to export the WAR file.

Export the SWTMessage proxy portlet
In a similar way, export the SWTMessage proxy portlet:

1. Right-click the com.ibm.itso.casample.swtmessage.portlet project and select Export -> Web -> WAR file.
2. Select the CASAMPLE folder as destination for the WAR file. Click Finish to export the WAR file.

Export the update site
The update site project must be exported so that it can be published. Follow these steps:

1. Right-click the com.ibm.itso.casample.updatesite project and select Export.
2. For the export destination select General -> File System and click Next.
3. Uncheck the .project file from the file list. Enter the CASAMPLE folder location in the To directory field and append \com.ibm.itso.casample.updatesite to it, as shown in Figure 15-26. Click Finish to export the update site.
4. Click Yes to create the directory if it is not already exists.
5. Verify that you have exported all the required artifacts. For this sample scenario, Figure 15-27 illustrates the exported artifacts in the CASAMPLE folder.

Figure 15-26  Exporting the update site

Figure 15-27  Exported portlets and update site
6. Since you have exported all the required artifacts, the Rational Application Developer development environment is no longer need it and you may want to close it.

15.3 Publishing the update site

Publishing the update site is as simple as copying the update file directory to the HTTP Server root document folder. Follow these steps to publish the ITSO Composite Application Sample update site:

1. Locate the HTTP Server root document folder. In this scenario, it is located at `c:\Program Files\IBM HTTP Server\htdocs\en_us`.

2. Copy the `com.ibm.itso.casample.updatesite` directory from the CASAMPLE folder to the HTTP Server document root directory as shown in Figure 15-28.

3. Verify the HTTP server is started. For example, in this scenario, the HTTP Server is started as a Windows service, so it can be verified by checking the service status, as shown in Figure 15-29.
4. Open an explorer window and enter the update site URL, for example http://localhost/com.ibm.itso.casample.updatesite. Figure 15-30 shows the update site published in the HTTP Server.

![Index of /com.ibm.itso.casample.updatesite](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Last modified</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parent Directory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>features/</td>
<td>19-Feb-2007 14:17</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>plugins/</td>
<td>19-Feb-2007 14:17</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>site.xml</td>
<td>19-Feb-2007 14:15</td>
<td>339</td>
<td></td>
</tr>
</tbody>
</table>

Figure 15-30 Update site published

5. Browse and review the published features, plugins and the site.xml file.
Assembling and deploying managed client composite applications

This chapter provides step-by-step instructions to assemble and deploy managed client composite applications for Lotus Expeditor V6.1 using WebSphere Portal and Network Client Install (NCI) tools.

This chapter provides the following topics:

- Creating a composite application template
- Assembling a composite application using Portal tools and NCI
- Adding portlets and RCP components to a composite application
- Defining roles in a composite application
- Instantiating a composite application from a template
- Assigning users to the composite application roles
- Provisioning the created composite application
- Running the composite application in Lotus Expeditor
16.1 Assembling the composite application

Assuming that you have already exported the components as WAR files and the update site has been published, you can now create and assemble the composite application using the Portal Tools. The steps required to create the composite application are:

- Install the portlet components (WAR files) into Portal
- Create users for the composite application
- Create the composite application template. In this task you will:
  - Add portlets to the composite application template
  - Wire portlets in the composite application
  - Define the Rich Client options for the composite application as well as options for each component.
  - In addition to the standard roles, create new roles as needed by the composite application
- Create a composite application instance based on the created template.
- Assign users to the roles defined for the composite application.

16.1.1 Installing portlets

Follow these steps to install the portlet components (WAR files) into Portal:

1. Log in into Portal as the Portal administrator user. For example, for a local portal server:
   a. Enter the URL: http://localhost:10038/wps/portal
   b. Login with user ID wpsadmin and password
2. Select Administration from the Launch menu.

SendMessage portlet

Install the SendMessage portlet as follows:

1. Click Install.
2. Enter the location of the SendMessage portlet WAR file, for example, directory CASAMPLE as shown in Figure 16-1, and click Next.
3. The wizard will show the portlet to be installed, see Figure 16-2. **Click Finish** to install the module.

4. A message indicates that the installation has been successful, as shown in Figure 16-3.

**ReceiveMessage portlet**

In a similar way, follow these steps to install the ReceiveMessage portlet:

1. Click **Install** to install the ReceiveMessage portlet
2. Enter the location of the ReceiveMessage portlet WAR file and click Next.
3. The wizard will show the portlets to install. Click Finish to install the module.
4. A message indicates that the installation has been successful.

**SWTMessage stub portlet**
In a similar way, follow these steps to install the SWTMessage stub portlet:

1. Click Install to install the SWTMessage stub portlet.
2. Enter the location of the SWTMessage stub portlet WAR file and click Next.
3. The wizard will show the portlets to install. Click Finish to install the module.
4. Verify that the SendMessage, ReceiveMessage and the SWTMessage stub modules are now installed. See Figure 16-4.

![Manage Web Modules](image)

*Figure 16-4  Installed portlets*

### 16.1.2 Adding a new user for the application

In this section you will create a user to test the roles for the application. Follow these steps:

1. Make sure you are logged into Portal as the Portal administrator user and select Administration from the Launch menu.
2. Select **Access - > Users and Groups** and click **New user**.

3. Enter **itsouser** as user ID and password. Enter other details as shown in Figure 16-5. Click **OK** to create the user.

![New user enrollment. Provide the information requested](image)

* User ID:
  * itsouser
* Password:
  * *********
* Confirm Password:
  * *********
* First Name:
  * ITSO
* Last Name:
  * User
* Email:
  * 
* Preferred language:
  * - Nothing Selected -

**Figure 16-5**  Adding a new user

4. The user will be created as shown in Figure 16-6. You will use this user to login and run the composite application.
16.1.3 Create a new template

In this section you will create a composite application template that will be used to create composite application instances to run in Lotus Expeditor. Follow these steps to create the template:

1. Log in as the Portal administrator user.
2. Select Templates from the Launch menu.
3. Go to the Template Library tab. Notice that by default the NCI installer creates a sample Expeditor MyClient Sample Template, as shown in Figure 16-7. The ITSO Composite Application Sample Template will be based on this template. Click New.
4. Enter ITSO Composite Application Sample template as the template name, select MyClient as the template category and select Expeditor My Client Sample Template as the starting point for the template, as shown in Figure 16-8.

**Note:** The category for this composite application template should be MyClient for Lotus Expeditor rich client desktops. Category for WebSphere Portal is Composite Applications.
5. Click **OK** to create the template.

### 16.1.4 Editing the template

Follow these steps to edit the created composite application template:

1. The template entry has been created and is listed in the Application Template Library, as shown in Figure 16-9. Click the template menu and select **Edit Template Layout**.

![Application Template Library](image)

*Figure 16-9 Application templates*

2. Change the default title for the application page. Click the Edit Page properties icon, it looks like a paper sheet as shown in Figure 16-10.
3. Enter ITSO Composite Application Sample as the page title, as shown in Figure 16-11 and click **OK** to save the changes.

4. Click the Edit Layout button, it looks like a pencil as shown in Figure 16-12.
5. In the Edit Layout page, click **Add** portlets to add the application portlets.

6. Select **Title contains** in the Search By field, enter **Message** in the Search box and click **Search**. Select the portlets shown in Figure 16-13 and click **OK**.

7. Reorganize the portlets using the layout buttons (arrows) so the layout looks as shown in Figure 16-14.
Wiring portlets

In this section you will create the wires as required by the composite application. See Figure 16-15 for details.

Select the **Wires** tab and execute the following steps to create the required wires:
1. SendMessage to ReceiveMessage wire. Enter the following values as shown in Figure 16-16:
   - Source portlet: SendMessage
   - Sending: wire_text
   - Target page: ITSO Composite Application Sample (unique name)
   - Target portlet: ReceiveMessage
   - Receiving: MessageWireAction,wire_text
   - Wire Type: Public
   - Click the add button (looks like a plus sign) to add the new wire

   **Note:** The wire type Public indicates that this wire is for all users.

![Figure 16-16  SendMessage to ReceiveMessage wire](image)

2. SendMessage to SWTMessage wire. Enter the following values as shown in Figure 16-17:
   - Source portlet: SendMessage
   - Sending: wire_text
   - Target page: ITSO Composite Application Sample (unique name)
   - Target portlet: com.ibm.itso.casample.swtmessage.portlet
   - Receiving: ReceiveMessageAction,receive_text
   - Wire Type: Public
   - Click the add button

![Figure 16-17  SendMessage to SWTMessage wire](image)

3. SWTMessage to ReceiveMessage wire. Enter the following values, as shown in Figure 16-18, and click the add button.
   - Source portlet: com.ibm.itso.casample.swtmessage.portlet
   - Sending: send_text
– Target page: ITSO Composite Application Sample (unique name)
– Target portlet: ReceiveMessage
– Receiving: MessageWireAction, wire_text
– Wire Type: Public
– Click the add button

Figure 16-18  SWTMessage to ReceiveMessage wire

Rich Client layout administration
In this section you will define the Rich Client options for the application and application components. Go to the Rich Client tab to invoke the Rich Client Administration portlet.

SendMessage portlet options
Execute the following steps:

1. Scroll down and click the Edit button of the SendMessage component, as shown in Figure 16-19.

Figure 16-19  Available portlets

2. Feature requirements. Define the client projection for the SendMessage component. Click Add under the Feature Requirement section, as shown in Figure 16-20.
3. Enter the following details, as shown in Figure 16-21.
   a. Feature Id: `com.ibm.itso.casample.sendmessage.feature`
   b. Feature version: `1.0.0`
   c. Matching rule: `compatible`
   d. Provisioning URL:
      `http://localhost/com.ibm.itso.casample.updatesite/`
   e. Click **Done** to finish.
4. The feature is associated to the SendMessage portlet. See Figure 16-22. Notice that the Lotus Expeditor client will try to find the plug-in components at the specified provisioning URL when the application is installed on the client.

![Figure 16-22 Added SendMessage feature requirements](image)

5. Instance description. Enter Send Message Component as the component instance description, as shown in Figure 16-23.

6. Select **This portlet runs locally on the rich client (requires client bundle)** and enter `/com.ibm.itso.demoapp.sendmessage.portlet` in the Portlet context root field, as shown in Figure 16-23.

![Figure 16-23 Rich client properties for SendMessage portlet component](image)

7. Layout properties. Scroll down to the Layout properties section, check **Add this view as standalone** and click **OK** to finish, as shown in Figure 16-24.
8. You will see the SendMessage component configured as shown in Figure 16-25.
**ReceiveMessage portlet options**

Execute the following steps:

1. Click the Edit button for the ReceiveMessage component.

2. Feature requirements. You will define the client projection for the ReceiveMessage component. Click **Add** under the Feature Requirement section.

3. Enter the following details, as shown in Figure 16-26.
   a. Feature Id: com.ibm.itso.casample.receivemessage.feature
   b. Feature version: 1.0.0
   c. Matching rule: compatible
   d. Provisioning URL:
      ```
      http://localhost/com.ibm.itso.casample.updatesite/
      ```
   e. Click **Done** to finish

![Feature requirement for ReceiveMessage portlet component](image)

*Figure 16-26  Feature requirement for ReceiveMessage portlet component*

4. The feature is associated to the ReceiveMessage portlet. See Figure 16-23.
5. Instance description. Enter Receive Message Component as the component instance description, as shown in Figure 16-28.

6. Select **This portlet runs locally on the rich client (requires client bundle)** and enter /com.ibm.itso.demoapp.receivemessage.portlet in the Portlet context root field, as shown in Figure 16-28.

7. Layout properties. Scroll down to the Layout properties section, check **Add this view as standalone** and click **OK** to finish.

8. You will see the ReceiveMessage component configured as shown in Figure 16-29.
SWTMessage stub portlet

Execute the following steps:

1. Click the Edit button to configure the SWTMessage component.

2. Feature requirements. You will define the client projection for the SWTMessage component. Click Add under the Feature Requirement section.

3. Enter the following details, as shown in Figure 16-30.
   a. Feature id: com.ibm.itso.casample.swtmessage.feature
   b. Feature version: 1.0.0
   c. Matching rule: compatible
   d. Provisioning URL:
      http://localhost/com.ibm.itso.casample.update/site/
   e. Click Done to finish.
4. The feature is associated to the SWTMessage portlet, as shown in XXX.

5. Instance description. Enter SWT Message Component as the component instance description, as shown in XXX.

6. Leave the option This portlet represents an SWT view on the rich client selected. Notice the value for the Eclipse view id, this value is taken by the tool from the SWTMessage stub portlet parameter, as shown in Figure 16-32.
7. Scroll down to the Layout properties section, check **Add this view as standalone** and click **OK** to finish.

8. You will see the SWTMessage feature requirements configured as shown in Figure 16-33.

9. In the Page properties section, select **Visible** and check **Add to the launcher** to define the application launch properties, as shown in Figure 16-34.
10. Click **OK** to make the changes effective.

11. Scroll down to the bottom of the page and click **Done** to exit from the Rich Client Administration portlet.

12. Click **Save** to save all the definitions made to the application template.

### 16.1.5 Defining the application roles

In this section you will define the required application user roles. Follow these steps:

1. Select **Manage Roles** from the application menu, as shown in Figure 16-35.
2. As illustrated in Figure 16-36, there are two roles defined by default:
   - Administrators. This role has all permissions assigned. They can edit the application and control membership
   - Users. This role does not have any assigned permissions.
3. Click **Users** to modify the Users role.
4. In this sample scenario, the Users role will provide access to the composite application portlet components.
   
a. For both, the SendMessage and the ReceiveMessage components select User (Users are allowed to view portal content) for the Level of Access field, as shown in Figure 16-37.
   
b. For the RCP component select the option No access (cannot access the portlet)
   
5. Click OK to save the changes.

![Figure 16-37 Component access settings](image)

6. Create a new role to be used by the composite application. In the Roles Administration portlet, click New.
7. Enter the following values:
   a. Select Users in the Create a new role based in field. This will copy the permissions from the Users role.
   b. Enter RCPUsers as the name for the new role
   c. Enter Users with access to RCP components as the role description
   d. For the SWTMessage component, select User (Users are allow to view portal content) in the Level of Access field, as shown in Figure 16-39. In this scenario, this role will allow access to the RCP component.
   e. Click OK to accept the changes.
8. Verify that the new role has been added, as shown in Figure 16-40.

9. Click **Save** to save the changes. When done, you will exit the Roles Administration portlet.
Saving a backup copy of the created template

In this section you will export the created template to create a backup copy. It is recommended and a best practice to backup the application template to an external file for later use.

1. From the template menu select Export Template as shown in Figure 16-41.
2. In the File Download window, click **Save** and enter a location for the template XML file and then click **OK** to save the file.

**Note:** Later, you can use this file to quickly recreate the template. For example, in cases where the template gets damaged or corrupted.

### 16.1.6 Creating an application instance

Once the application template has been created, you will need to create an application instance based on this template. Follow these steps:

1. Select **Templates** from the Launch menu.

2. Go to the **My Client Applications** tab. There are no applications defined at this time, see Figure 16-42. You will create a new application based on the previously created template. Click **New**.

![Figure 16-42 Defining a new application](image)

3. As illustrated in Figure 16-43, enter the following values:
   a. Application name: ITSO Composite Application Sample
   b. Application template: ITSO Composite Application Sample template
   c. Click **OK** to create the application.
4. Wait while the application is created. A message will indicate that the application has been successfully created, as shown in Figure 16-44.

5. Click **ITSO Composite Application Sample** to preview the composite application running on WebSphere Portal, as shown in Figure 16-45.
Assign users to the application roles

The composite application instance has been create and you will now need to assign existing users to the available application roles. Follow these steps:

1. Click the application menu and select Assign Application Members, as shown in Figure 16-46.
Figure 16-46 Assigning application users

2. The application roles are displayed as well as users assigned to a specific role. Notice that the application owner, wpsadmin in this case, is assigned to the Administrator role, as shown in XXX.

Figure 16-47 Assigning users to applications roles

3. Select **Add** → **Add Users** to add members to the Users role.
4. In the Directory Search window, enter ITSO into the **Search for:** box and click **Search**. See Figure 16-48.

5. In the Search results box, select ITSO User and click **Add** to add the user to the Users application role, as shown in Figure 16-48.

![Figure 16-48  Assigning users to the Users role](image)

6. Click **OK** and verify that the ITSO user has been added to the Users role, as shown in Figure 16-48.
7. Click **Done** to finish.

### 16.2 Deploying and running the composite application

Now that the composite application has been created using the Portal Tools and the Network Client Installer (NCI), you will now deploy the composite application from the created Update Site and run the application in Lotus Expeditor desktop.

**Important:** Make sure you have a network connection between Lotus Expeditor Client and WebSphere Portal.

Figure 16-50 illustrates how the Lotus Expeditor Client platform downloads the composite application XML from WebSphere Portal. The information from this XML file is used to access the Update Site and download the application components for execution.
Follow these steps:

1. Open your Lotus Expeditor Client. If asked enter the KeyStore password and click **OK**.
2. From the menu bar select **File → Preferences**.
3. Select **Home Portal Account** and enter the following details, as shown in Figure 16-51.
   a. Description: Home Portal Account
   c. Name: itsouser
   d. Password: itsouser
   e. Authentication URL: http://localhost:10038/wps/j_security_check
   f. Authentication Type: J2EE-FORM
Chapter 16. Assembling and deploying managed client composite applications

4. Click **OK**. Click **Yes** to the warning about updating the Expeditor configuration.

5. Select **Portal Applications** from the Open menu. Note the ITSO Composite Application Sample is listed, as shown in Figure 16-52.

6. Right-click ITSO Composite Application Sample and select **Open**. Notice that because the **ITS0 User** is assigned to the Users role, he only has access to the portlet components, as shown in Figure 16-53.
7. Enter a message in the SendMessage portlet component and click **Submit**. The message is received by the ReceiveMessage portlet, as shown in Figure 16-53.

### 16.2.1 Changing roles

In this section, you will update and modify the role of the ITSO User from Users to RCPUsers. Follow these steps:

1. Login to Portal as Administrator, wpsadmin in this case.
2. Go to **Launch -> Templates -> My Client Applications**.
3. Click ITSO Composite Application Sample.
4. From the application menu select **Assign Application Members** as shown in Figure 16-54.
5. Select **Actions -> Manage Members** as shown in Figure 16-55.

![Figure 16-55  Manage users](image)

6. Select the ITSO User and click **Reassign** as shown in Figure 16-56.

![Figure 16-56  Reassigning a role](image)
7. In the **Reassign members to** field select **RCPUsers**, as shown in Figure 16-57. Click **OK**.

![Figure 16-57  Selecting the new role](image)

8. The ITSO user is now assigned to the RCPUsers role as shown in Figure 16-58. This new role will provide access to the RCP component in addition to the portlet components. The RCPUsers role was defined in 16.1.5, “Defining the application roles” on page 420.
Chapter 16. Assembling and deploying managed client composite applications

9. Click **Done** and then **Done** again to finish.

**Refreshing the composite application in Lotus Expeditor**

In this section you will refresh and run the composite application in Lotus Expeditor Client. The composite application will include the updated role and will provide access to the RCP component. Follow these steps:

1. Open your Expeditor Client.
2. From the **Open** menu, select **Portal Applications**.
3. Right-click ITSO Composite Application Sample and select **Open** to refresh the composite application.
4. Because ITSO User is now in the RCPUsers role, he or she will have access to the SWT Message component, as shown in Figure 16-59, “Composite application with RCP component” on page 438.
5. Enter a message in the SendMessage portlet and click **Submit**. The message is displayed in the ReceiveMessage portlet and in the SWTMessage component, as shown in Figure 16-60.

---

**Figure 16-59**  Composite application with RCP component

**Figure 16-60**  RCP component receiving a message from the property broker
6. Enter a message in the SWTMessage component and click Send. The message is displayed in the ReceiveMessage portlet as shown in Figure 16-61.

![Image of SWT component sending a message to the property broker]

**Figure 16-61  RCP component sending a message to the property broker**
Adding components to a composite application

This chapter illustrates how a composite application template can be reused to create slightly different composite applications in order to satisfy specific user needs. This chapter provides step-by-step instructions to create an application instance, assemble a new layout, add a new component and deploy the new managed client composite application in Lotus Expeditor V6.1.

The following topics are described in this chapter:

- Installing a portlet in WebSphere Portal for remote access
- Instantiating a new composite application from an existing template
- Adding a new component (a remote portlet) to the instantiated composite application
- Updating the application roles
- Assigning users to the application roles
- Deploying and running the new composite application in Lotus Expeditor client
17.1 Installing the weather remote portlet

In this sample scenario you will install and publish a sample remote portlet that shows the local weather. Follow these steps to add the weather portlet to WebSphere Portal:

1. Download the weather portlet WAR file from additional materials.
2. Log in to WebSphere Portal as administrator user, wpsadmin for example.
3. Select Launch → Administration.
5. Enter the location of the weather portlet WAR file as shown in Figure 17-1 and click Next.

![Figure 17-1 Installing a portlet in Portal for remote access](image)

6. Click Finish to install the Web Module.
7. Select Portlet Management -> Portlets.
8. Select Title contains in the Search by field, enter weather in the Search field and click Search. The weather portlet appears as shown in Figure 17-2.
Figure 17-2  Installed portlet

9. Click the Provide portlet icon (it looks like a star) as shown in Figure 17-3. Click OK in the confirmation dialog.

10. The portlet is now provided as a remote portlet, as shown in Figure 17-3.

Figure 17-3  Portlet provided for remote access

17.2 Creating a new composite application instance

In this section you will create a new application instance from a template. You will then add a remote portlet component to this application by using the Remote Portlet Viewer, provided by the Network Client Installer (NCI).

Follow these steps:

1. Select Templates from the Launch menu.
2. Go to **My Client Applications** tab. Click **New**.

3. Enter ITSO Composite Application + Remote Portlet Sample as the application name and select ITSO Composite Application Sample template as the application template, as shown in Figure 17-4.

4. Click **OK** to create the application instance.

![Creating a new composite application instance](image)

5. Wait until the application is created. A message will indicate the application has been successfully created.

6. Click **ITSO Composite Application + Remote Portlet Sample**.

7. From the application menu, select **Edit Application Layout**, as shown in Figure 17-5.
8. Click the Edit Page Properties icon shown in Figure 17-6.

9. Enter ITSO Composite Application + Remote Portlet Sample as the page title, as shown in Figure 17-7 and click OK.
10. Click the Edit layout icon shown in Figure 17-8.

11. Click **Add Portlets** under the SendMessage component, as shown in Figure 17-9.
12. Select **Title Contains**, enter WSRP in the search box and click **Search**. See Figure 17-10.

13. Select the **Rich Client WSRP Viewer portlet** as shown in Figure 17-10 and click **OK**.

14. Verify that the portlets are layout as shown in Figure 17-11.
15. Click **Done** and **Done** again to finish.

**Configure the WSRP Viewer**
In this section you will configure the Rich Client WSRP Viewer to show the weather portlet. Follow these steps:

1. Right-click the upper right corner of the portlet to open the Rich Client WSRP Viewer menu.

2. Select **Configure** as shown in Figure 17-12.

![Figure 17-12](image)

**Figure 17-12** Select option to configure the WSRP Viewer

3. In the WSDL address enter the Portal Remote Portlet location, as shown in Figure 17-13, and click **OK**.

![Figure 17-13](image)

**Figure 17-13** Remote portlet WSDL location

4. Click **com.ibm.itso.compapp.carrental.weather**, as shown in Figure 17-14, to select the weather portlet as the remote portlet to be displayed in the Viewer.
5. Select **Don’t need a Clone**, as shown in Figure 17-15, and click **OK**.

6. The portlet is now configured to show the weather portlet as shown in Figure 17-16.

**Updating the application roles**

In this section you will update the composite application roles. Follow these steps:
1. From the page menu select Manage Application Roles as shown in Figure 17-17.

![Figure 17-17  Manage application roles](image)

2. In this scenario, you will grant access of this Rich Client WSRP Viewer to users assigned to the RCPusers role. Click **RCPUsers** as shown in Figure 17-18.

![Figure 17-18  Selecting the application role](image)

3. In the Rich Client WRSP Viewer, select **User (Users are allowed to view portal content)** under Level of Access, as shown in Figure 17-19, and click **OK**.
Figure 17-19  Users are allowed to view portlet content

4. Click **Done** to finish.

**Assigning users to application roles**

The composite application instance has been created, you will now need to assign existing users to the application roles. Follow these steps:

1. Click the application menu and select **Assign Application Members**, as shown in Figure 17-20.
2. The application roles are displayed. Users assigned to these roles are also displayed. Notice that the application owner, wpsadmin in this case, is assigned to the Administrator role, as shown in Figure 17-21.

3. Select Add → Add RCPUsers. In the Directory Search window, enter ITSO in the Search for box and click Search. In the Search results box, select the
ITSO User and click **Add** to add the user to the RCPUsers application role, as shown in Figure 17-22.

![Figure 17-22  User added to the RCPUsers application role](image)

4. Click **OK**. You will see the ITSO user added to the RCPUsers application role, as shown in Figure 17-23.
17.3 Deploying and running the application

In this section you will deploy and run the composite application in Lotus Expeditor. Follow these steps:

1. Open your Lotus Expeditor Client. If asked enter the KeyStore password and click Login.

2. Select Portal Applications from the Open menu. Press F9 to update the application list. Notice that the ITSO Composite Application + Remote Portlet Sample is listed, as shown in Figure 17-24.
Chapter 17. Adding components to a composite application

3. Right-click **ITSO Composite Application + Remote Portlet Sample** and select **Open**. The new weather component has been added to the application as shown in Figure 17-25.

Figure 17-24  New application instance is listed

![Table showing application instances](image)

Figure 17-25  Components in composite application
Multipage applications and cross-page wiring

This chapter provides information about how to assemble managed composite applications using multipage capabilities and cross-page wiring. A sample application is included to illustrate the required steps when wiring components assembled in different pages.
18.1 Overview

In this chapter we describe how to create a multipage composite application based on an existing composite application template. Once the composite application has been assembled, we show you how to wire components residing in different pages.

There are some considerations you should take into account when developing multi-page composite applications and they are:

- Actions for target components must be declared as Global.
- Cross-page wires only can be created between two pages. For example, if you have wired a component on page 1 to a second component on page 2, any other components deployed in page 1 can only be wired to portlets on page 2.

18.2 Creating a multipage composite application

In this section you will create an application instance based on the ITSO Composite Application Sample template and you will customize it to have more than one page. Also you will see how to define wires that cross pages.

The first page in the application will contain the SWTMessage and the ReceiveMessage components and the second page will contain the SendMessage portlet component.

Follow these steps to create the composite application:

1. Log in into portal as an administrative user, for example log in as wpsadmin.
2. Select Templates from the Launch menu.
4. Enter ITSO Composite Application CrossWire Sample as the application name and select ITSO Composite Application Sample template as the application template, as shown in Figure 18-1. Click OK to create the application.
5. Wait until the application is created. A message will indicate that the application has been successfully created.

6. Click **ITSO Composite Application CrossWire Sample**.

7. From the application menu, select **Edit Application Layout**, as shown in Figure 18-2.

8. Click the Edit Page Properties icon for the page, as shown in Figure 18-3.
9. Enter ITSO CA - First Page as the page title, as shown in Figure 18-4 and click OK.

10. Click the Edit layout icon shown in Figure 18-5.
11. For the first page you will remove the SendMessage portlet and leave the other components untouched. Click the SendMessage menu and select **Delete portlet**, as shown in Figure 18-6.

12. Click **OK** in the confirmation box. Figure 18-7 illustrates how the new page layout will look after deleting the SendMessage component.

13. In order to receive properties from other pages, the receiving components, SWTMessage and ReceiveMessage in this case, have to declare their actions as Global.
14. Go to the Wire tab and click Manage Actions, as shown in Figure 18-8.

![Figure 18-8 Selecting the Manage Actions option](image)

15. Mark all the receiving portlet actions as Global, as shown in Figure 18-9, and click OK.

![Figure 18-9 Setting global actions](image)
16. Click **Done** to exit the Edit layout mode for the first page.

17. Next, you will create a second page for the application. In the Application Layout portlet, click **New Page**.

18. Enter **ITSO CA - Second Page** as the page title, as shown in Figure 18-10, and click **OK**.

![Page Properties](image)

*Figure 18-10  Second page in the composite application*

19. Click the Edit Page Properties icon for the page, as shown in Figure 18-11.
20. Click **Add portlets** to add the SendMessage portlet.

21. Select **Title contains** in the Search by field, enter *message* in the Search field and click **Search**. The SendMessage portlet should appear as shown in Figure 18-12. Select **SendMessage** and click **OK**.

22. The SendMessage portlet is added to the Second Page. You will now add the required wires to send a message to the components in the first page. Click the **Wires** tab. Enter the following details and click the add wire button (looks like a plus sign) as shown in Figure 18-13.

   - **Source portlet:** SendMessage
   - **Sending:** wire_text
   - **Target page:** ITSO CA - First Page
   - **Target portlet:** ReceiveMessage
23. Add a second wire by entering the following details and clicking the Add Wire icon as shown in Figure 18-14.

- Source portlet: SendMessage
- Sending: wire_text
- Target page: ITSO CA - First Page
- Target portlet: com.ibm.itso.casample.swtmessage.portlet
- Receiving: ReceiveMessageAction, receive_text
- Wire Type: Public

24. You will also need to specify the Rich Client options for the SendMessage portlet. Go to the Rich Client tab. Click the Edit button of the SendMessage component, as shown in Figure 18-15.
25. You will define the client projection for the SendMessage component. Click **Add** under the Feature Requirement section, as shown in Figure 18-16.

![Feature requirements](image1)

*Figure 18-16 Adding feature requirements*

26. Enter the following details, as shown in Figure 18-17. Click **Done** to finish.
   a. **Feature Id:** `com.ibm.itso.casample.sendmessage.feature`
   b. **Feature version:** `1.0.0`
   c. **Matching rule:** `compatible`
   d. **Provisioning URL:**
      `http://localhost/com.ibm.itso.casample.updatesite/

![Configure feature requirements](image2)

*Figure 18-17 Configure feature requirements*

27. The feature will be associated to the SendMessage portlet, as shown in Figure 18-18. Note that the Lotus Expeditor client will look for the plug-in
components at the provisioning URL when the application is installed on the client.

28. Enter Send Message Component as the component instance description. Select **This portlet runs locally on the rich client (requires client bundle)** and enter `/com.ibm.itso.demoapp.sendmessage.portlet` in the Portlet context root field, as shown in Figure 18-19.

29. Scroll down to the Layout properties section, check **Add this view as standalone** and click **OK** to finish, as shown in Figure 18-20.
30. In the Page properties section, make sure **Visible** is selected and click **OK**, as shown in Figure 18-21.
31. Scroll down to the bottom of the page and click **Done** to exit from the Rich Client Administration portlet.

32. Back into the Application Layout portlet, click **Inactive** under the Status field for the Second Page to switch the status to **Active**. Click **OK** in the confirmation window.
33. Click **Done** to save the changes.

34. From the page menu, select **Manage** Application Roles as shown in Figure 18-23.

![Manage Application Roles](image)

*Figure 18-23  Manage associated application roles*

35. For example, grant access to the SendMessage portlet in the Second Page to RCPusers. Click RCPUsers as shown in Figure 18-24.

![RCPUsers Role](image)

*Figure 18-24  RCPUsers role*

36. For the SendMessage component, select **Users** under Level of Access, as shown in Figure 18-25, and click **OK**.
18.2.1 Assigning users to application roles

Now that the multipage application instance is created, you will need to assign existing users to the application roles. Follow these steps:

1. Click the application menu and select **Assign Application Members**, as shown in Figure 18-26.
2. Select **Add → Add RCPUsers**. In the Directory Search window, enter ITSO into the **Search for:** field and click **Search**. In the Search results box, select ITSO User and click **Add** to add the user to the application role, as shown in Figure 18-27.
3. Click **OK**. You will see the ITSO user added to the RCPUsers role, as shown in Figure 18-28.

![Figure 18-28 Added user to RCPUsers role](image)

4. Click **Done** to finish.

### 18.2.2 Running the application in Lotus Expeditor

Now that the multi-page application has been defined using the Portal tools, you will deploy and run the application using Lotus Expeditor. Follow these steps:

1. Open your Lotus Expeditor Client. If asked enter the KeyStore password and click **OK**.
2. Select **Portal Applications** from the Open menu.
3. Push F9 to update the application list. The ITSO Composite Application CrossWire Sample application should appear.
4. Right-click ITSO Composite Application CrossWire Sample and select **Open**, as shown in Figure 18-29.
5. You will see a selector bar allowing you to switch between pages. Note that ITSO CA - First page is the current page, as shown in Figure 18-30. Enter some text in the SWTMessage component and click **Send**. As you can see, wires in the same page are still working.

6. Select **ITSO CA - Second Page** from the page selector. The SendMessage component is displayed, as shown in Figure 18-31. Enter some text in the SendMessage component and click **Submit**.
7. The message will be received and displayed by the components in the first page as shown in Figure 18-32.
The ITSO Car Rental sample composite application

This chapter contains step-by-step instructions on how the ITSO Car Rental sample application was developed. First, it gives you an introduction to the basic architecture of the application and eventually leads to the sample steps about how to deploy the ready application to a WebSphere Portal for provisioning to Lotus Expeditor client desktops.
19.1 Architectural overview

The design goal for the Composite Applications sample application was to create a scenario that is simple and does not require special knowledge to comprehend. On the other hand, the application must bare some complexity to allow the use of various components, which must communicate between each other.

The sample application is a Car Rental scenario, which is an application that can be used by representatives in a car rental firm when booking rental cars for customers.

It is clear to the authors of this book, that such an application will, in a real world scenario, be well beyond the scope of this document. It is also clear that a real world application will contain many other features with regard to transaction safety, user security, and so on, which are also beyond the scope of this book. In other words, our intent is to create something that is just a mock up example but useful enough to demonstrate the concepts of composite applications.

Infrastructure complexity has been reduced to the absolute minimum. We do not use messaging, nor transactions or any other Java 2 Platform, Enterprise Edition (J2EE) mechanisms. In order to be able to deal with a couple of records, a simple IBM Cloudscape™ database with three tables is used.

As this is a managed application, we also want to demonstrate a feature of role based administration. Hence, two scenarios are built. One for the representative booking the cars, and one for the supervisor who can look up customers and see customer history information that the representative cannot see.

The incident that must be covered by the representative part of this application is described as follows:

1. A customer calls in to reserve a car.
2. The representative enters the customer number and issues a search for the customer.
3. Once the customer is found:
   a. The car selection component is updated to list the cars of the preferred size of this customer.
   b. The credit card component is updated to show the customer’s credit card information.
4. The representative selects a car in the car selection component, which makes the booking component update its section for car details.
5. The representative enters the start and end dates for the rental period, the booking component automatically updates the total price for this rent.
6. The representative issues a credit card authorization request.

7. The credit card component authorizes or rejects the requests and returns the result to the booking component.

8. Only if the booking component receives a credit card authorization does it enable the *Book*, which is the button, to allow the final booking of the car.

The diagram for the composite application as seen by the car rental representative is shown in Figure 19-1.

![Diagram of the ITSO Car Rental for the representative role](image)

*Figure 19-1  Diagram of the ITSO Car Rental for the representative role*

Additionally, the scenario for the supervisor is as follows:

1. The supervisor looks up a customer.

2. Once the customer is found:
   
   a. The car selection component is updated to list the cars of the preferred size of this customer.

   b. The contract list component shows a history of bookings by this customer.
This diagram for the composite application as seen by the supervisor is illustrated in Figure 19-2.

![Diagram of the ITSO Car Rental for the supervisor role]

**19.2 The database component**

This component is a helper that deals with customers, cars and contract records throughout the whole application. It provides the three bean types representing the record types and also handles reading and writing to and from the Cloudscape database.

In its activator class, the component checks for the existence of the database in the user’s Lotus Expeditor home directory. If it is not available, it creates and initially populates the directory.

**19.2.1 Customer model**

This section describes the customer table and data bean.
Customer record
The statement shown in Example 19-1 is used to create the customer table.

Example 19-1  SQL-Statement to create the customer table

```sql
CREATE TABLE "CUSTOMERS"
(
    "ID" CHAR(4) NOT NULL,
    "FNAME" VARCHAR(10),
    "LNAME" VARCHAR(10),
    "STREET" VARCHAR(30),
    "CITY" VARCHAR(20),
    "ZIP" VARCHAR(5),
    "STATE" CHAR(2),
    "PHONE" VARCHAR(20),
    "EMAIL" VARCHAR(30),
    "PREF" CHAR(1),
    "CCTYPE" CHAR(2),
    "CCNUM" VARCHAR(20),
    "PHOTO" BLOB(64K),
    PRIMARY KEY(ID)
);
```

Customer bean
The bean com.ibm.itso.compapp.carrental.database.Customer can be used to represent such a record. It provides getters and setters for the fields listed in Table 19-1.

Table 19-1  Customer bean fields, getters and setters

<table>
<thead>
<tr>
<th>Column name</th>
<th>Bean field</th>
<th>Getter</th>
<th>Setter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>String id</td>
<td>getId</td>
<td>setId</td>
</tr>
<tr>
<td>FNAME</td>
<td>String fname</td>
<td>getFirst</td>
<td>setFirst</td>
</tr>
<tr>
<td>LNAME</td>
<td>String lname</td>
<td>getLast</td>
<td>setLast</td>
</tr>
<tr>
<td>STREET</td>
<td>String street</td>
<td>getStreet</td>
<td>setStreet</td>
</tr>
<tr>
<td>CITY</td>
<td>String city</td>
<td>getCity</td>
<td>setCity</td>
</tr>
<tr>
<td>ZIP</td>
<td>String zip</td>
<td>getZip</td>
<td>setZip</td>
</tr>
<tr>
<td>STATE</td>
<td>String state</td>
<td>getState</td>
<td>setState</td>
</tr>
<tr>
<td>PHONE</td>
<td>String phone</td>
<td>getPhone</td>
<td>setPhone</td>
</tr>
</tbody>
</table>
19.2.2 Car model

This section describes the car table and data bean.

**Car record**
The statement shown in Example 19-2 is used to create the car table.

*Example 19-2  SQL-Statement to create the car table*

```sql
CREATE TABLE "CARS"
(
   "ID" CHAR(4) NOT NULL,
   "MAKE" VARCHAR(20),
   "MODEL" VARCHAR(20),
   "RATE" DOUBLE,
   "COLOR" VARCHAR(10),
   "SIZE" CHAR(1),
   "SHIFT" VARCHAR(10),
   "PHOTO" BLOB(64K),
   PRIMARY KEY(ID)
);```

**Car bean**
The bean com.ibm.itso.compapp.carrental.database.Car can be used to represent such a record. It provides getters and setters for the fields shown in Table 19-2.

*Table 19-2  Car bean fields, getters and setters*

<table>
<thead>
<tr>
<th>Column name</th>
<th>Bean field</th>
<th>Getter</th>
<th>Setter</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
<td>String id</td>
<td>getId</td>
<td>setId</td>
</tr>
<tr>
<td>MAKE</td>
<td>String make</td>
<td>getMake</td>
<td>setMake</td>
</tr>
</tbody>
</table>
19.2.3 Contract model

This section describes the contract table and data bean.

Contract record
The statement shown in Example 19-3 is used to create the contract table.

Example 19-3  SQL-Statement to create the contracts table

```
CREATE TABLE "CONTRACTS"
(
   "ID" INT GENERATED ALWAYS AS IDENTITY (START WITH 1),
   "CUSTID" CHAR(4) NOT NULL,
   "CARID" CHAR(4) NOT NULL,
   "DTSTART" DATE,
   "DTEND" DATE,
   "SLOT" CHAR(2),
   "AMOUNT" DOUBLE,
   PRIMARY KEY(ID)
);
```

Contract bean
The bean com.ibm.itso.compapp.carrental.database.Contract can be used to represent such a record. It provides getters and setters for the fields shown in Table 19-3.

<table>
<thead>
<tr>
<th>Column name</th>
<th>Bean field</th>
<th>Getter</th>
<th>Setter</th>
</tr>
</thead>
<tbody>
<tr>
<td>MODEL</td>
<td>String model</td>
<td>getModel</td>
<td>setModel</td>
</tr>
<tr>
<td>RATE</td>
<td>Double rate</td>
<td>getRate</td>
<td>setRate</td>
</tr>
<tr>
<td>COLOR</td>
<td>String color</td>
<td>getColor</td>
<td>setColor</td>
</tr>
<tr>
<td>SIZE</td>
<td>String size</td>
<td>getSize</td>
<td>setSize</td>
</tr>
<tr>
<td>SHIFT</td>
<td>String shift</td>
<td>getShift</td>
<td>setShift</td>
</tr>
<tr>
<td>PHOTO</td>
<td>byte[] photo</td>
<td>getPhoto</td>
<td>setPhoto</td>
</tr>
</tbody>
</table>

Table 19-3  Contract bean fields, getters and setters
19.2.4 Database helper functions

To access the database, the component provides static functions that can be called to retrieve records from the database and also to write records to the booking database:

- **getCustomersByID**
  
  Retrieves an array of customer records that satisfy the provided search ID which can be a wildcard in the SQL notation. For example, 12% to find all customers starting with 12. The method syntax is:

  ```java
  public static Customer[] getCustomersByID(java.lang.String id)
  ```

  - Parameters
    - ID or wildcard ID
  - Returns
    - An array of all customers found or an array of zero size if either no customers were found or an exception occurred

- **getCarsBySize**

  Retrieves an array of car records that satisfy the provided search ID, which can be a wildcard in the SQL notation. For example, 12% to find all cars starting with 12. The method syntax is:

  ```java
  public static Car[] getCarsBySize(java.lang.String size)
  ```

  - Parameters
    - ID or wildcard ID
  - Returns
    - An array of all cars found or an array of zero size if either no cars were found or an exception occurred

<table>
<thead>
<tr>
<th>Column name</th>
<th>Bean field</th>
<th>Getter</th>
<th>Setter</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARID</td>
<td>String carid</td>
<td>getCarid</td>
<td>setCarid</td>
</tr>
<tr>
<td>DTSTART</td>
<td>Date dtstart</td>
<td>getDtstart</td>
<td>setDtstart</td>
</tr>
<tr>
<td>DTEND</td>
<td>Date dtend</td>
<td>getDtend</td>
<td>setDtend</td>
</tr>
<tr>
<td>SLOT</td>
<td>String slot</td>
<td>getSlot</td>
<td>setSlot</td>
</tr>
<tr>
<td>AMOUNT</td>
<td>Double amount</td>
<td>getAmount</td>
<td>setAmount</td>
</tr>
</tbody>
</table>
19.3 The booking component

The booking component is the controlling part of the application. The plug-in is kept to a minimum complexity.

BookingActivator
This is the bundle’s activator class that is registered with the bundle in its MANIFEST.MF file as shown in Example 19-6 on page 487.

Access to the bundle’s context
At a later time, you will require access to the bundle’s context to retrieve another class that you need for letting the composite communicate with other components. The bundle’s context is required to retrieve this information; therefore, when the platform calls your bundle start() method with the context as this method’s parameter, you store the context value for future retrieval. Example 19-4 shows the added start() method.
Example 19-4  Changes to the bundle’s activator class

...  

    // The shared context
    private static BundleContext context;

...

public void start(BundleContext context) throws Exception  
{
    this.context = context;
    super.start(context);
}
...

ImageDescriptor
As the plug-in has an image in its Standard Widget Toolkit (SWT) composite, a function has been added for conveniently retrieving a descriptor for this image. The function is illustrated in Example 19-5.

Example 19-5  Retrieving an image descriptor for the booking plug-in

...

/**
 * Returns an image descriptor for the image file at the given plug-in
 * relative path
 *
 * @param path
 * the path
 * @return the image descriptor
 */
public static ImageDescriptor getImageDescriptor(String path)  
{
    return imageDescriptorFromPlugin(PLUGIN_ID, path);
}

BookingComposite
From the user interface perspective, the BookingComposite has three parts:

- The customer information part
- The car information part
- The booking information part
Figure 19-3 illustrates the three sections or parts in the BookingComposite component.

![Booking Component](image)

**Figure 19-3  Picture of the booking component**

**MANIFEST.MF**

The MANIFEST.MF file holds the bundle’s core information. Notice that it requires the `com.ibm.rcp.propertybroker.swt` bundle and imports the package `com.ibm.pvc.topology`.

**Example 19-6  Booking component's MANIFEST.MF file**

```manifest
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Booking Plug-in
Bundle-SymbolicName: com.ibm.itso.compapp.carrental.booking;singleton:=true
Bundle-Version: 1.0.0
Bundle-Vendor: IBM Corporation
Bundle-Localization: plugin
Require-Bundle: org.eclipse.ui,
  org.eclipse.core.runtime,
  com.ibm.rcp.propertybroker,
  com.ibm.itso.compapp.carrental.database,
  com.ibm.rcp.propertybroker.swt
```
Eclipse-LazyStart: true
Export-Package: com.ibm.itso.compapp.carrental.booking
Import-Package: com.ibm.pvc.topology

**plugin.xml**
The only extension that the component declares is the single view that it has. Example 19-7 shows the content of the plugin.xml file.

**Example 19-7   Booking component's plugin.xml file**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?eclipse version="3.2"?>
<plugin>
  <extension
    point="org.eclipse.ui.views">  
    <view
      allowMultiple="true"
      category="ITSO"
      class="com.ibm.itso.compapp.carrental.booking.BookingView"
      id="com.ibm.itso.compapp.carrental.booking.BookingView"
      name="%main.view.name">  
      </view>
    </extension>
  </plugin>
```

### 19.4 The car selection component

The car selection component is a basic portlet (JSR168) implementing the Model View Control (MVC) design pattern. MVC is concerned with separation of responsibilities. The objective, no matter how it is applied or what type of application, is to segregate a system into components. Each component should be small, identifiable, self-contained and reusable. These components are
identified by the role they play in the system. Each role in the system may have several classes working in conjunction to achieve a common goal.

![Select your car size:](image)

<table>
<thead>
<tr>
<th>Photo</th>
<th>Manufacturer</th>
<th>Model</th>
<th>Rate</th>
<th>Color</th>
<th>Shift</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Car Photo" /></td>
<td>Brown</td>
<td>Lima</td>
<td>421.0</td>
<td>Silver</td>
<td>Automatic</td>
</tr>
<tr>
<td><img src="image" alt="Car Photo" /></td>
<td>Yellow</td>
<td>Munich</td>
<td>399.99</td>
<td>Black</td>
<td>Automatic</td>
</tr>
</tbody>
</table>

*Figure 19-4  Car Selection portlet*

**Model**

The model component is responsible for encapsulating all the business logic required by the component. It must be independent of other components in the system. To achieve this objective, it must be able to retrieve data required to complete the business rules and accept very generic receive parameters. The database class's responsibility is to interact with databases and obtain the required data. In this sample portlet component, the method getCarsBySize() retrieves all cars that satisfy the user's preferred car size. The sample code is listed in Example 19-8.

*Example 19-8  Model component*

```java
public static Car[] getCarsBySize(String size) {
    Car[] cars = new Car[0];

    try {
        Connection conn = createConnection();
        Statement st = conn.createStatement();

        ResultSet rs = st.executeQuery("SELECT COUNT(*) FROM CARS WHERE SIZE LIKE '\"' + size + '\"'");
        rs.next();
        int records = rs.getInt(1);

        rs = st.executeQuery("SELECT * FROM CARS WHERE SIZE LIKE '\"' + size + '\"'");
        cars = new Car[records];
        int a = 0;
        while (rs.next()) {
            cars[a++] = new Car(rs.getString("Make"), rs.getString("Model"), rs.getDouble("Rate"), rs.getString("Color"), rs.getString("Shift"));
        }
    }
```
cars[a] = new Car(rs);
a++;
}
if (conn != null) conn.close();
} catch (SQLException e) {
    e.printStackTrace();
}

return cars;

Controller
At the heart of the MVC architecture is the controller component. First, it must evaluate the validity of the request, including the portlet state and mode, and any parameter information passed as part of the request. The controller component then decides which model component provides the required functionality to satisfy the business requirements of the request. The basic portlet (JSR168 API) controller is used to carry out the interaction with other components and involves a two-step process (processAction and doView methods).

View
The doView() method is called by the Expeditor portlet container when the portlet is in view mode. It is intended to contain logic to display the portlet View page. Example 19-9 shows the doView() method in the sample portlet component.

Example 19-9  Method doView()

```java
public void doView(RenderRequest request, RenderResponse response)
    throws PortletException, IOException {
    response.setContentType(request.getResponseContentType());
    request.setAttribute("carList", carList);
    request.setAttribute("wirePreferedCarFromBooking",
        wirePreferedCarFromBooking);
    PortletRequestDispatcher rd =
        getPortletContext().getRequestDispatcher(
            getJspFilePath(request, VIEW_JSP));
    rd.include(request, response);
}
```

The setAttribute() method is used to bind objects to the current page. Objects exist for the duration of the page and are shared between multiple pages. In this sample portlet component, there are two variables:

- `carList` is an array from the database that lists all values from the car table.
wirePreferredCarFromBooking is a property received from the property broker with the selected car.

The processAction() is called when the user submits a request to a portlet, for example, to process an input from a user action. Example 19-10 shows the processAction() method.

**Example 19-10  processAction() method**

```java
public void processAction(ActionRequest request, ActionResponse response) throws PortletException, java.io.IOException {
    wirePreferredCarFromBooking = request.getParameter(WIRE_PREFERRED_CAR) == null ?
        request.getParameter("selTypeCar");
    request.getParameter(WIRE_PREFERRED_CAR);

    if (request.getParameter("selCarH") == null ||
        request.getParameter("selCarH").equals("")) {
        carList = Database.getCarsBySize(wirePreferredCarFromBooking);
    } else {
        int selcar = Integer.parseInt(request.getParameter("selCarH"));
        request.setAttribute("Car Make", carList[selcar].getMake());
        request.setAttribute("Car Model", carList[selcar].getModel());
        request.setAttribute("Car Size", carList[selcar].getSize());
        request.setAttribute("Car Color", carList[selcar].getColor());
        request.setAttribute("Car Slot", carList[selcar].getSlot());
        request.setAttribute("Car Rate", new Double(carList[selcar].getRate()).toString());
    }
}
```

When the processAction() method receives a request, it checks if the action comes from the property broker or from a JavaServer Pages (JSP) page submit. If the action came from the property broker, it must get the car type (selTypeCar); otherwise, it gets the last selected option in the menu, since the user has the option to choose other car types.

The processAction() method also checks if the user has selected a rental car. The JSP page saves the rental car in the selCarH property. When the property is null, it indicates the user has not yet selected a car; otherwise, the JSP sends the vector position to indicate the selected car.

After a car has been selected, the processAction() method matches the values from the property broker to the values from the database. The name in the action must be the same value configured in the WSDL file.
The JSP component is responsible for creating a presentation resource to display the results of the model part. The same applies for other MVC parts, the View part must be independent of the other components in the system. Its success or failure must not depend on the success or failure of the model. In practice, several different view components may be developed in order to create a dynamic, complete and possibly multi-purpose application. In a typical portlet environment, the view is created using JSPs. In this sample project, the name of the JSP is CarPortletView.jsp. The sample JSP header is shown in Example 19-11.

Example 19-11  JSP Header

```jsp
<% 
    PortletURL actionUrl = renderResponse.createActionURL();
    actionUrl.setParameter(PortSelectCar.ACTION_NAME_PARAM,
    PortSelectCar.SEND_MESSAGE_ACTION);

    Car[] carList = (Car[]) renderRequest.getAttribute("carList");
    String wirePreferedCarFromBooking =
    (String)renderRequest.getAttribute("wirePreferedCarFromBooking");
%>
```

The JSP form is shown in Example 19-12.

Example 19-12  JSP Form

```jsp
<form method="POST" name="frmCar" action="<%= actionUrl.toString() %>">
</form>
<br />
<%if(wirePreferedCarFromBooking.equals("C"))
out.println("selected"); %>
<option value="C">Compact</option>
<%if(wirePreferedCarFromBooking.equals("M"))
out.println("selected"); %>
<option value="M">Medium</option>
<%if(wirePreferedCarFromBooking.equals("F"))
out.println("selected"); %>
<option value="F">Full</option>

<input name="<%=com.ibm.itso.compapp.carrental.choosecar.PortSelectCar.FORM_SUBMIT%>" type="submit" value="OK"
</div>
```
19.5 The credit card component

In this section the credit card component is described.
19.6 The customer contract list component

The customer contract list component is a J2EE Web component using MVC design pattern.

The model component is responsible for encapsulating all the business logic required by the component. It must be independent of the other components in the system. The database class is responsible for interacting with the database and obtaining the proper data. The method getContractsByCustID() gets all contracts from a customer. Example 19-13 shows the model portion of the component.

Example 19-13 Model component

```java
public static Contract[] getContractsByCustID(String id) {
    Contract[] contracts = new Contract[0];
    try {
        Connection conn = createConnection();
        Statement st = conn.createStatement();
        ResultSet rs = st.executeQuery(
            "SELECT COUNT(*) FROM CONTRACTS
            WHERE CUSTID LIKE '\''+ id + '\''");
        rs.next();
        int records = rs.getInt(1);
        rs = st.executeQuery("SELECT a.ID, CUSTID, CARID, DTSTART, DTEND,
            SLOT, AMOUNT, FNAME, LNAME, MODEL FROM CONTRACTS a, CUSTOMERS b, CARS c
            WHERE a.CUSTID LIKE '\''+ id + '\'' AND a.CUSTID = b.ID AND a.CARID = c.ID"");
        contracts = new Contract[records];
        int a = 0;
        ```
```
Figure 19-5 Customer contract list component
```java
while (rs.next() && a < records) {
    contracts[a] = new Contract(rs);
}
if (conn != null) conn.close();
}
```}

```java
return contracts;
}
```

The `getContractsByCustID()` method receives the customer ID and selects all contracts for this customer from the database.

The controller component evaluates the validity of the request, including the user’s state and any parameter information passed on as part of the request. The controller component then decides which Model component provides the functionality to satisfy the business requirements of the request. The controller component is represented by a servlet.

The method `doGet()` is called when the page is loaded. This method is responsible for forwarding the request to render the Web page. The vector has a list of all contracts. In this case, the property is empty since the user is still loading the page. Example 19-14 shows the `doGet()` method.

```
Example 19-14  doGet method

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
    Contract[] list = new Contract[0];
    request.setAttribute("contractList", list);
    request.getRequestDispatcher("/view/ContractList.jsp").forward(request, response);
}
```

The `doPost()` method is responsible for obtaining the customer ID from the request and verifies all contracts available in the database. After doing this, results are sent to the JSP. Example 19-15 illustrates the `doPost()` method.

```
Example 19-15  doPost method

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

```
Contract[] list =
Database.getContractsByCustID(request.getParameter("txtCustomerNumber"));
    request.setAttribute("contractList", list);
}
request.getRequestDispatcher("/view/ContractList.jsp").forward(request, response);

The JSP page is responsible for creating a presentation resource for the results of the Model component. As in all MVC components, the View must be independent of the other components in the system. In the sample project, the name of JSP is ContractList.jsp.

Example 19-16  JSP Page

<jsp:useBean id="contractList" class="java.lang.Object"
    scope="request"></jsp:useBean>

<form
    action="/com.ibm.itso.compapp.carrental.listcontracts/ServContractList"
    method="post" name="frmContract">
    <table border="0">
        <tbody>
            <tr>
                <td style="font-family: Arial;"
                    width="150">Customer Number</td>
                <td width="33">
                    <input type="text" name="txtCustomerNumber"
                        size="10" maxlength="10"></td>
            </tr>
            <tr>
                <td width="150">
                    <input type="submit" name="cmdOK" value="OK">
                    <input type="reset" value="Clear">
                </td>
                <td width="33"></td>
            </tr>
        </tbody>
    </table>
</form>

<table border="1" cellpadding="1" cellspacing="5">
    <tbody>
        <tr style="font-style: normal; font-family: Arial;" background-color: #80ffff; font-weight: 200; font-variant: normal">
19.7 The weather component

WebSphere Portal V6.0 provides support for the Web Services Remote Portlets (WSRP) standard enabling portal administrators, to provide portlets as WSRP services and integrate WSRP services as remote portlets. These remote portlets act in the same manner as they do for local JSR 168 portlets in the Consumer portal, regardless of how they are implemented on the producer portal.

In order to implement WSRP with WebSphere Portal, it is necessary to perform the following tasks:

- For producer portals:
  - Provide or withdraw a portlet

- For consumer portals:
  - Register an existing producer in the consumer portal
  - Consume a WSRP service provided by the producer
For purposes of this scenario, a WebSphere Portal V6.0 system has been configured to act as a portlet provider server and the Lotus Expeditor acts as a consumer of the remote portlet.

19.7.1 Tasks for producer portals

Producers provide portlets to be available remotely to portlet consumers. To provide or withdraw a portlet, portal administrators use the Manage Portlets administration portlet or use the XML configuration interface.

In this section, the common way to provide and withdraw a portlet in WebSphere Portal is explained. The sample weather portlet is selected to work as a remote portlet (WSRP portlet). Figure 19-6 illustrates the sample portlet.

Figure 19-6 Weather portlet

Note: The sample weather portlet shows a single JSP page and has no connections or communications with other components.

Providing a portlet using the administration portlet
This section describes how to enable the sample weather portlet for remote access:

1. Log in as an administrator user of the Producer portal.
2. Select Administration ➔ Portlet Management ➔ Portlets.
3. In the Manage Portlets portlet, search for the portlet you want to provide and click the **Provide portlet** icon as shown in Figure 19-7.

![Figure 19-7](image)

**Figure 19-7** Selecting a portlet to be provided

4. Click **OK** to provide the portlet. See Figure 19-8.

![Figure 19-8](image)

**Figure 19-8** Confirming the operation

5. A message appears indicating that the portlet has been successfully provided. As shown in Figure 19-9, a check mark appears in the Provided column of the portlet.

![Figure 19-9](image)

**Figure 19-9** Produced portlet for remote access
Withdrawing a portlet using the Manage Portlets portlet

To withdraw a portlet that has been previously provided, follow these steps:

1. Log in as an administrator user of the Producer portal.

2. Select Administration → Portlet Management → Portlets. In the Manage Portlets portlet, search for the portlet you want to withdraw and click Withdraw portlet. See Figure 19-10.

3. Click Yes to withdraw the portlet. A message appears indicating that the portlet has been successfully withdrawn.
Assembling Portal-administered composite applications

This chapter describes how to enable the Lotus Expeditor Client to receive the configuration information from a WebSphere Portal Server where the application is assembled.

In this chapter, the following topics are described:

- Client Composite Application Infrastructure (CAI)
- Network Client Installer (NCI)
- Assembling composite applications using NCI
20.1 Overview

In previous chapters we showed how to layout and assemble applications programmatically using Rational Application Developer, in this chapter we focus on using the portal admin tools to create the layout and wire the components of our application and then install the application onto the Lotus Expeditor Client using the Composite Application Portal catalog.

Lotus Expeditor V6.1 integrates a Composite Application Infrastructure (CAI) in order to deploy and execute composite applications at the client side. From the server side a new component named Network Client Installer (NCI) is required for supporting Lotus Expeditor.

In a portal administered environment we work with pages and portlets that we can compare to perspectives and SWT views, respectively, in the eclipse terminology. Portlets are actually a placeholder on the screen to display components in the client. What finally runs on the client are standard eclipse-based SWT views.

By projecting the views through WebSphere Portal, you can:

- Leverage portal policy, which exposes different application features to users based on their job responsibilities.
- Wire components together at runtime instead of at development time.
- Use a common deployment and aggregation model for both Web-based and client-based applications.

20.2 Composite Application Infrastructure

The Composite Application Infrastructure (CAI) at the client platform, is the composite application runtime environment which is needed in order to install and execute certain composite applications, in particular those applications composed in a WebSphere Portal Server environment.

To instantiate a composite application on the Lotus Expeditor client, for example, the client platform needs to know what are the components of that composite application, where to find those components, and how those components interact with each other. All this information, required in order to instantiate a composite application on the client side is encoded in an XML stream. This XML stream acts as a template and contains the necessary information to instantiate a composite application:

- The list of components which make up the application.
Chapter 20. Assembling Portal-administered composite applications

The layout of these components.

The links between components, the wires from the CA XML file will be translated to a Property Broker wire by the Topology Handler.

The roles and access permissions.

Other information required for the proper instantiation of the application.

The XML stream describing a composite application can be obtained from a static definition deployed to the local file system, a definition contained within a plug-in, or may also be retrieved from a portal server in a Portal-managed environment.

20.3 Network Client Installer

You can install Network Client Installer (NCI) on Expeditor Server, a standalone Web server, or a Portal server. In this section we describe the components deployed on Portal Server after the installation of NCI that we use later to assembly and deploy the composite application of the scenario.

20.3.1 Portlets installed by NCI

The following portlets are installed by NCI to allow the Lotus Expeditor client to be managed from a Portal Server:

- Workbench Administration portlet. This portlet enables WebSphere Portal to manage items contributed to the various portions of the Lotus Expeditor workbench.

  Figure 20-1 shows the Workbench Administration Portlets which are used to manage the eclipse based clients:

  - The shutdown, logoff, lock, password, task list, clock, and keyboard layout portlets are used to manage the eclipse platform.
  
  - Native application is used to manage the ability to run native desktop applications from the eclipse platform.
  
  - Custom eclipse extensions are used to manage preferences with XML on the eclipse platform.
Downloads portlet. This portlet allows a user to initiate the installation of the Lotus Expeditor client platform from the WebSphere Portal Server. This portlet is just a link to the download applet. By default, this portlet is not placed in any page so to use it, you need to place this portlet into a page.

Rich Client Layout Administration portlet. This portlet enhances WebSphere Portal Server in that it allows the application composer to specify options a properties for pages and page components of Rich Client composite applications. It is integrated in the WebSphere Portal Server's Page Customizer as a new tab titled **Rich Client** as shown in Figure 20-3. Recall that the Page Customizer contains portlets for editing the layout, content, and appearance of pages.
Policy Type Editor for Eclipse Preferences is an extension to the Resource Policies portlet so that you can specify eclipse preferences for the rich client based on policy types. It adds three new policy types:

- Client managed settings.
- Client connectivity preferences.
- Client portlet viewer settings.

In the Rich Client Templates page you can upload your own policy type as shown in Figure 20-4.
Rich Client Managed Browser Administration portlet acts as a placeholder to place a Web browser as a portlet, and lay it out on a portal page. Used by the administrator it allows to aggregate local or remote Web application within a portal-defined perspective.

Rich Client WSRP Viewer enablement portlet is used to put a WSRP instance on the rich client. Figure 20-5 shows the configure mode of this portlet.

Rich Client View Placeholder portlet acts as a placeholder for an Eclipse view and can be placed on a page. This portlet has the following sections:

- Rich client properties. Here you specify if the portlet represents an eclipse view or a portlet view.
– Feature requirements. Here you specify prerequisite features that the client will need before starting the target view.
– Layout properties. Additional parameters needed to display the eclipse view since it could be more granular and complex than Portal.

Rich Client View Placeholder

<table>
<thead>
<tr>
<th>Properties shown on this page apply to only the instance of this portlet on this page.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configure rich client properties for Rich Client View Placeholder</td>
</tr>
</tbody>
</table>

**Instance description:**

- **Rich client properties**
  - **Type of view or extension:** simple view
  - **This portlet represents an SWT view on the rich client**
    - **Eclipse view id:**

- **This portlet runs locally on the rich client (requires client bundle)**

- **Portlet context root**

- **Feature requirements** Specify the features required for this component on the rich client.

<table>
<thead>
<tr>
<th>Feature ID</th>
<th>Feature version</th>
<th>Matching rule</th>
<th>Provisioning URL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>There are no entries to display.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Layout properties**

*Figure 20-6  Rich Client View Placeholder portlet*

### 20.3.2 Templates and applications

NCI also installs an application template named **Expeditor MyClient Sample Template** and place it in a new category called **MyClient**. When you create a new application using this template, the template automatically creates a Workbench Contribution page. Note that this page is marked as hidden by default, so you need to mark it as visible or create another page to place your content if you want to contribute to the Expeditor Client user interface.
Building Composite Applications

20.4 Sample scenario

In previous chapters we explained how to assemble and deploy composite applications in Lotus Expeditor in a programmatic way. Now we are going to create a new composite application using the declarative approach working in a Portal-managed environment. We will use the Portal tools to assemble and deploy the composite application.

Lotus Expeditor V6.1 integrates a Composite Application Infrastructure (CAI) to deploy and execute composite applications at the client side. As a requirement for the server side a component named Network Client Installer (NCI) must be installed in WebSphere Portal Server.

The composite application of the scenario contains the following components:

- The Customer Identification portlet
- Customer Details component, an Eclipse SWT view

We define roles to access the components. A user with role of users, for example an employee of the call center who just inform customer about their reservations, cannot access to the customer details component because it contains customer’s
credit cards information. Only a user with supervisor role can gain access to this kind of information.

This scenario describes the following issues:

► Create a composite application in an abstract way using templates. This template defines the components, wirings and behave of the components based on roles.
► Portlets will be mapped to actual portlets (that run in the local container) and to an Eclipse based SWT view.
► Create an instance of the composite application from our template. The composite application is stored in a catalog on the server.
► The client will make the instantiation of the composite application. Portal responds with an XML document which describes the instructions to create the entire application. The client will ask Portal for provisioning of the missing components and copy them into a client component library.

20.4.1 Preparing the application for deployment

In this section the preliminary tasks required to assemble and deploy the sample composite application are described.

Creating the placeholder JSR168 portlet

To register properties and actions of the SWT component using CAI we need to create a blank JSR168 portlet which contains the WSDL file of the component.

1. From Rational Application Developer, select File → New → Other, and under Client Services section, select Client Services Portlet Project.
2. In the New Client Services Portlet Project enter the following values:
   a. Project name: com.ibm.itso.compapp.carrental.customerdetails.portlet
   b. Target Runtime: Client Service v6.1
   c. Portlet API: JSR 168 Portlet
   d. Portlet name: com.ibm.itso.compapp.carrental.customerdetails.portlet
   e. Portlet type: Empty portlet.
   f. Click Next.
3. In the Portlet Settings window change the package prefix to com.ibm.itso.compapp.carrental.customerdetails.portlet and the class prefix to CustomerDetailsPortlet. Click Finish.
4. Expand the newly created project and in the WebContent directory create a new folder called **wsdl**. Copy the CustomerDetails.wsdl file from com.ibm.itso.compapp.carrental.customerdetails project to this folder.

5. Update the portlet deployment descriptor to include a reference to the WSDL file. Open the portlet.xml and add the code shown in Example 20-1 under the `<portlet-info>` section.

   **Example 20-1   Updating portlet.xml file**

   ```xml
   <portlet-preferences>
   <preference>
     <name>com.ibm.portal.propertybroker.wsdllocation</name>
     <value>/wsdl/CustomerComponent.wsdl</value>
     <read-only>true</read-only>
   </preference>
   </portlet-preferences>
   ```

6. Save and close the file.

7. Now export the project. Select the project and select **Export → Web → WAR file**

8. In the WAR Export window enter a name and a destination for the project. Make the name similar to the SWT component, for example com.ibm.itso.compapp.carrental.customerdetails.portlet. Click Finish.

**Export the com.ibm.itso.compapp.carrental.userreservations**

We also need to export the com.ibm.itso.compapp.carrental.userreservations client services portlet project as a WAR file. Select the project and click **Export → Web → WAR file.** Enter the name of the Web module and the destination and click **Finish.**

**Note:** We modified the display names of the portlets before exporting the project. Now the portlets begin with XPD plus its name.

**Attention:** You need to export this project and import it again in Portal Server in order to modify the WSDL files to have the same namespace as the ITSO application. As an alternative, you could reuse the portlets from Portal scenario because these portlets are essentially a placeholder on the screen to display components in the client and they register properties and actions with Portal Property Broker to establish wires between components.

**Creating the required feature projects**

To deploy plug-ins into an update site you need to create a feature project for each plug-in.
Before creating a Feature project for com.ibm.itso.compapp.carrental.customerdetails plug-in open the plugin.xml and in the Extensions tab ensure that the value of allowMultiple field is true for the ITSO customer details view as shown in Figure 20-8.

![Image of Extensions tab showing allowMultiple set to true](image)

**Figure 20-8  Allow multiple in views.**

Select the Build tab an ensure that the plugin references files are exported during a binary build.

![Image of Binary Build and Source Build settings](image)

**Figure 20-9  Binary Build for Customer Details component.**
Save and close the plugin.xml file.

Follow these steps to create the Feature projects:

1. Select File → New → Other in the menu bar. Select Feature Project under Plug-in Development section and click Next.

2. In Feature Properties window enter
   com.ibm.itso.compapp.carrental.customerdetails.feature as the feature project name, review the feature properties and click Next.

3. In the next window select the

4. The feature.xml file will be opened. Go to the Dependencies tab and include the com.ibm.itso.compapp.carrental.database plug-in and feature. Click the Compute button. This action adds the required plug-ins and features for your feature. You should have a list similar as shown in Figure 20-10.

5. In the Included Features tab include the

6. Save and close the file.

7. Repeat the steps to create a new Feature project for
   com.ibm.itso.compapp.carrental.userreservations plug-in.
      a. Call the project
         com.ibm.itso.compapp.carrental.userreservations.feature
      b. Include the com.ibm.itso.compapp.carrental.userreservations plug-in.
      c. In Dependencies tab of the feature.xml file Compute the dependencies.
d. Save and close the feature.xml file.

8. Create a new Feature project for com.ibm.itso.compapp.carrental.database:
   a. Call the project com.ibm.itso.compapp.carrental.database.feature
   b. Include the com.ibm.itso.compapp.carrental.database plug-in.
   c. In Included Features tab add the org.apache.derby.feature
   d. In Dependencies tab of the feature.xml file Compute the dependencies.
   e. Save and close the feature.xml file

9. Create a new Feature project for com.ibm.itso.compapp.carrental.synchronize
   a. Call the project com.ibm.itso.compapp.carrental.synchronize.feature
   b. Include the com.ibm.itso.compapp.carrental.synchronize plug-in.
   c. In Included Features tab add the org.apache.derby.feature,
      com.ibm.mobileservices.isync.db2j.feature and
      com.ibm.mobileservices.isync.feature
   d. In Dependencies tab of the feature.xml file Compute the dependencies.
   e. Save and close the feature.xml file.

Creating the update site project
Lastly you create an update site where features can be deployed. This site will be referenced by the assembled composite application XML file on the Portal Server. For example:

1. From Rational Application Developer select File → New → Other in the menu bar.

2. Select Update Site Project under Plug-in Development section and click Next.

3. In the Update Site Project window enter the project name, for example com.ibm.itso.compapp.carrental.userreservations.updatesite and click Finish.

4. The site.xml file from the newly created update site project will be opened. In the Site Map tab we have to add the features created before. Create new categories and add the features as shown in Figure 20-11. Select Build All to compile features and plug-ins.
514 Building Composite Applications

5. Save and close the file.

6. Export the update site by right-clicking the project and select Export. Then select General → File System and click Next.

7. Select the project and enter a directory location to export the resources and click Finish.

20.4.2 Setting up the application in Portal

In this section you will find the steps required to set up the sample application in WebSphere Portal.

Install portlets

Now we are ready to import the portlets into Portal Server.

1. Log in Portal Server and select Administration from the Portal Launch menu.

2. Select Portlet Management → Web Modules and click Install

3. Browse to the directory where you exported the war files and click Next.
4. The next window shows the content of your WAR file, review it and click the Finish button to install the WAR file.

5. Repeat the steps to install com.ibm.itso.compapp.carrental.userreservations.war file.

6. As the portlet’s names are the same as in Portal scenario if you have already installed those portlets in Portal Server rename the english portlets titles of the new ones to make portlets search easier. To do so, click the Web module, then click the portlet application and in the next window for each portlet do the following:
   a. Click the spanner icon and then select I want to set titles and descriptions. Select the English locale and click the pencil icon, and then change the title and click OK.
b. Click **OK** two times more to go back to portlets list.

c. Repeat the steps with the other two portlets.

**Copy update site to HTTP Server directory**

Copy the update site you exported in “Creating the update site project” on page 513 to a directory in the HTTP Server you can access. For example: “<HTTPServer_path>/htdocs/en_US” where <HTTPServer_path> is the directory where your HTTP Server is installed.

You should now be able to browse your site via a Web browser as shown in Figure 20-14.

![Figure 20-14 Accessing update site via a Web browser](image)

**Note**: This is the URL you will use to download the features in CAI.

### 20.4.3 Assembling the application

In this section the sample composite application assembly process is described.

**Creating the template**

To start assembling the application we are going to create a template which describes our application in an abstract way.

1. Log into Portal Server as administrator and select **Templates** from the Portal Launch menu. The Welcome page for Templates appears.

2. Select **Template Library** page and click **New** button to create a new template. Enter the following values as shown in Figure 20-15.
a. Template name: ITSO Car Rental Reservations Template
b. Category: MyClient
c. Starting point: Expeditor MyClient Sample Template, this template is basically a blank canvas and is added to Portal Server when you installed the Network Client Installer.

![Application Template Library](image)

**Figure 20-15 Creating new template.**

3. Click **OK** to create the template.

**Editing the template layout**

In this section the composite application template layout is illustrated. Follow these steps:

1. From the Application Template Library select the arrow besides the newly created template to launch the context menu and click **Edit Template Layout**.

2. As we selected Expeditor MyClient Sample Template as starting point a page called Workbench Contribution has been created. Renamed this page by selecting Edit Page Properties icon as shown in Figure 20-16. Enter ITSO User Reservations as the new title.
Here you can add more pages and labels in the same way as you usually do in a portal page content.

**Placing portlets on ITSO User Reservations page**

Follow these steps to design the application page layout.

1. Edit the ITSO User Reservations page by clicking the pencil icon.
2. In the **Content** tab of Page Layout screen click **Add portlets** and select the portlets we installed in “Install portlets” on page 514.
Figure 20-17  Adding portlets to a page.

3. Reorganize your portlets as shown in Figure 20-18.

Figure 20-18  Moving portlets

**Configuring the portlets**

In this section the portlet components are configured. For example:

1. After the portlets are placed on the page we need to edit the portlet instance preferences. Go to the **Rich Client** tab where the Rich Client Layout Administration Portlet is located. This portlet is used to manage common rich client-specific metadata. This portlet handles the parameters that are common to all perspectives and views.
2. In the first section you configure the Page properties. Select **Visible** radio button to make the perspective accessible. Here you can also specify if you want to add your application to the launcher in the client user interface etc. For further information about these properties refer to the Lotus Expeditor information center.

![Figure 20-19  Defining Page Properties](image)

3. Click **OK** to save your changes.

4. Scroll down to reach the list of the portlets in the current page as shown in Figure 20-20.
5. Clicking the edit icon of each portlet takes you to a page to modify the portlet properties. These properties will be stored as portlet instance data for this page.

6. Clicking the edit icon for **XPDUserReservationsPortlet** takes you to the page to modify its properties. Go to the features requirement section where you can specify the Eclipse features that are needed as prerequisites to render this portlet properly on the client. To do so, click **Add** button.

7. The page to edit properties of a feature appears. Enter the following values as shown in Figure 20-21 and then click **Done**.
   a. Feature ID: `com.ibm.itso.compapp.carrental.userreservations.feature`
   b. Feature version: `1.0.0`
   c. Matching rule: this field defines the version filtering algorithm used by the Eclipse update manager. Select **compatible**
      Refer to the Lotus Expeditor information center for information about matching rule types.
      This field specifies the URL where the client fetches for required features and it is the same that you used to access the update site via a Web browser in Figure 20-14 on page 516.
8. It returns to the XPDUserReservationsPortlet properties page.
   a. Enter an instance description of this portlet, for example Customer Search
   b. Select the radio button This portlet runs locally on the rich client.
   c. Enter the portlet context root: /UserReservationsProject
      The portlet context root is defined in the plugin.xml file of the project.
9. Scroll down to the layout properties section and uncheck **Show the title bar** option to display the view without the title bar.
10. Click **OK** to save the changes and it returns to the page that contain the portlet list of the page.

11. Repeat the same steps you did to configure the properties of XPDUserReservationsPortlet with the other two portlets: XPDCustomerReservations and XPDReservationDetails. They require the same feature and have the same context root. Enter Customer Reservations and Reservation Details as instance description respectively.

12. Lastly click the edit icon of the com.ibm.itso.compapp.carrental.customerdetails.portlet to modify its properties. Add a new feature entering the following values:
   a. Feature id: com.ibm.itso.compapp.carrental.customerdetails.feature
   b. Feature version: 1.0.0
   c. Matching rule: compatible
   d. Provisioning URL:
13. In the instance description field, enter Customer Details, and in the Rich client properties section, select the radio button This portlet represents a SWT view on the rich client. Enter the eclipse view ID com.ibm.itso.compapp.carrental.customerdetails.CustomerView. The view ID is defined in the plugin.xml file, refer to Figure 20-8 on page 511.

![Rich Client Layout Administration](image)

Figure 20-24 Entering eclipse view ID

14. In the layout properties section, uncheck Show the title bar option and click OK to save the configuration.

**Wiring components**

In this section, we describe the process to wire components using the Portal tools.

1. Select the Wires tab, and from the drop-down boxes, select each option (see Figure 20-25 on page 526).

2. Select public as wire type and then click the plus button to add the wire to the portal page.
3. Return to the **Content** tab and click **Done**. You are returned to the Application Layout page.

4. Select **Save** button to save the template and it returns to the Template Library page.

**Managing template roles**

In this section, application roles associated to the sample composite application template are created. For example:

1. From the Portal Launch menu select Template and then select Template Library page.

2. Select the arrow besides the ITSO Car Rental Reservations Template to launch the context menu and click **Manage Roles**.

3. Click **New** to create a new role. Create a new role called **Supervisor** and based on the **User** Role, in the Components Access Settings section, give the level of access shown in Figure 20-26. Users with this role have privileged user rights to all the components.
Figure 20-26  Creating a new role

4. Click OK to return to the window with the list of all roles where you can see your new Supervisor role.

5. Now we are going to modify the Users role. Select Users role from the list and in the next window modify the level of access of the components as shown in Figure 20-27. Users with this role cannot access to the customer details component.
6. Click **OK** to save the role and then click **Save** to return to the Template Library page.

**Creating new users in Portal**

To show the roles behavior you need to create two new users in Portal Server. To do so follow these steps:

1. Select **Administration** from the Portal Launch menu.
2. Select **Access → Users and Groups**. Click **New User** to create supervisorITSO. Provide the information needed and click **OK**.
3. Repeat the steps to create user ITSO user.

Creating an application library

In this section, the sample composite application is instantiated based on the created template. Follow these steps:

1. Select MyClientApplications tab and then the New button to create a new application based on the ITSO Car Rental Reservations template.
2. Click **OK** and it returns to MyClient Application tab where you can see your new application. Click it to edit the application.

3. A page with the four portlets we added to the template appears. Select **Assign Application Members** from the page’s context menu to open the membership component of the application.

4. In the Members window, select the arrow besides **Add button** and click **Add Supervisor**, Figure 20-30 on page 531 appears. Enter ITSO in the search field and click **Search**. The left panel displays the results of the query. In this panel select **ITSO Supervisor** and click **Add >** button to display it in the right panel. Click **OK**.
5. Repeat the previous step selecting Add Users instead of Add Supervisor and in the Directory Search window select ITSO User from the left panel to add it to the right panel. Click OK. You will see the following users by role as shown in Figure 20-31 on page 532.
6. Click **Done**.

### 20.4.4 Deploying the composite application

Now start the Lotus Expeditor Client to install the application from Portal Server.

1. Select **File → Preferences** and in the Preferences window select **Home Portal Account**.
2. You will be prompted to confirm if you want to update your portal account configuration. Click Yes.

3. Open the Portal Applications Catalog. Select Open → Portal Applications. In the Categories window select MyClient category under Composite Applications section. You should see the ITSO Car Rental Reservations application on the right. If you cannot see the application select View → Refresh.
4. Open the ITSO Car Rental Reservations by double-clicking. You should see a perspective as shown in Figure 20-34.

5. Modify the user account to userITSO and open the application again. Figure 20-35 on page 535 shows the application appearance as user role.
Note: Before running the application as user, ensure that the ITSORentals Cloudscape database is created in the client. As a supervisor, the customer details component creates this database if it is not already created, but as user, none of the components do it.
Component development tools

This chapter provides an overview of different development tools for component development. It is aimed toward the component developer who needs to develop NSF or Eclipse components as well as portlets.
21.1 Lotus Domino Designer 8

A major new enhancement in Lotus Notes 8 is the support for composite applications. Using Lotus Domino Designer 8 you can extend Notes applications to be NSF components in a composite application. A composite application can include any combination of NSF and Eclipse components as well as local portlets.

This chapter gives you a brief introduction to the enhanced composite application support of Lotus Domino Designer 8.

Before you can use the Lotus Domino Designer 8 you have to make sure it is installed on your system and that you do have sufficient access rights on the Notes applications that you want to edit. When you install Lotus Notes 8 you need to make sure to select the check box as shown in Figure 21-1.

![Select Domino Designer during installation](image)

Figure 21-1 Select Domino Designer during installation

Development responsibilities for building composite applications can be distributed across several types of application development and administration team members. The process does not have to be restricted to the highly skilled component developer. The roles in composite application development typically include the following:

- Component developer
  - Design and creation of NSF and Eclipse components.
- Application assembler
Definition and assembling of the composite application. May be a line of business user.

- Application administrator
  Deployment of portlets onto WebSphere Portal and maintaining NSF-based composite applications on the Domino server.

Lotus Domino Designer 8 is targeted at the Notes application and NSF component developer

### 21.1.1 The user interface

Once you have opened a new Notes application using **File** → **Application** → **Open**, Lotus Domino Designer contains two main areas an application developer can utilize when editing an application as shown in Figure 21-2.

- Designer Pane
  In the left pane you have bookmarks to your Notes applications. If you click a bookmark of a Notes application all the design elements show up. Here you can also see a new design element for composite applications.

- Developers Pane
  Depending on the design element selection the developers view opens on the right side of the workbench. This contains details for the specific design element you are currently working with.
There are no significant changes to the user interface compared to previous versions. The changes are in the available design elements and support for composite applications.

### 21.1.2 Composite application design elements

There are a number of new design elements related to composite applications which are new in this release. You can find them in the designer pane of Lotus Domino Designer. There you have access to the following design elements that you need to use when creating and editing NSF based composite applications as well as NSF components.

- Properties

  This design element gives you access to the WSDL file within the Notes application. This WSDL file references data types, output properties, and actions available for each component in the composite application.
Applications

In NSF based composite applications, this design element gives you access to required XML file that consist of design notes that contain composite application definitions for that composite application. This XML is stored within the Notes application used for the composite application.

21.1.3 Property Broker Editor

A new extension to Lotus Domino Designer 8 is the Property Broker Editor. It can be launched from within a Notes application when you go to the new Properties design element.

In order for composite components to interact with each other data types, actions and properties have to be defined and stored in a WSDL file. This is a XML file and you can use the Property Broker Editor to create, edit and update it.

In Figure 21-3 you can see what the user interface looks like. There are three tabs that you can select at the bottom:

► Properties tab

Here you can define your properties that your NSF component should publish. You have the ability to define the name of a property as well as the type. In this tab you can also specify the target namespace.

► Types tab

If you do not want to use the default types, you need to add new types to your WSDL file. Make sure to specify the namespace for your new types here as well.

► Actions tab

In order for your NSF component to act upon properties published by other components you need to define WSDL actions that trigger Notes actions. You can specify a name for your new action and define either things like input parameters.

During the development of the use case scenario you will use the Property Broker Editor to create the WSDL file for your NSF component.
To make use of the data types, output properties, and actions available for each component in a composite application, you must import and store a copy of a WSDL file within the Notes application that references these elements for each component.

The WSDL files are stored in the properties design element as shown in Figure 21-4. There you have a range of different actions for handling WSDL files. This includes:

- New Properties
  It lets you create a new WSDL file.

- Import WSDL
  It lets you import an existing WSDL file from the file system.
Chapter 21. Component development tools

- **Open File**
  This opens the WSDL file stored in the Notes application with the new Property Broker Editor explained in 21.1.3, “Property Broker Editor”.

- **Open With**
  If you do want to open the WSDL file using another WSDL editor you can use this action button.

- **Export WSDL**
  Creates a copy of your WSDL file on the file system.

- **Merge WSDLs**
  Merge two or more WSDL files and the contained data types, properties and actions into one file.

- **Refresh**
  It lets you refresh the WSDL file.

*Figure 21-4  WSDL handling in Lotus Domino Designer 8*
21.2 Lotus Component Designer

Lotus Component Designer 6.0.1 (targeted for later in 2007) will start to deliver on its theme of develop once and run on multiple platforms. Component Designer 6.0.1 will expand its list supported platforms to include the Lotus Expeditor and Lotus Notes rich client platforms. This will allow customers to develop one application and deploy it to both WebSphere Portal server and to these rich client platforms. With Lotus Expeditor and Lotus Notes adding support for composite applications, developers can provide a similar experience for users of both Web browsers as well as rich clients. And, these Component Designer applications can be run off-line.

Components (aka applications) can be deployed to a WebSphere Portal server for subsequent provisioning to the client. Developers will be able to use the Lotus Notes 8 Composite Application Editor to add Component Designer components to a composite application. When an user accesses a composite application from the rich client, the components would be provisioned to that user's machine. This initial provisioning not only provisions the component to the rich client but it will also create the local XML document store. End users can then synchronize their data between the local and server XML document store using a pre-built synchronization service. Figure 21-5 on page 545 shows an example of an application build with Lotus Component Designer running in Lotus Notes 8.
Developers will use the same development approach as they do for Portal but they will provided additional capabilities allowing them to take advantage of the rich client. Lotus Component Designer will allow developers to take advantage of platform services like drag and drop, native dialogs and menus. They will also be allowed to run and debug their applications on a local rich client directly from the Lotus Component Designer IDE.

**Note:** You can check for the public beta of Lotus Component Designer 6.0.1 on the developerWorks® Lotus Component Designer site. Here is the link:

This chapter provides an overview of IBM Eclipse based Lotus Notes 8 client in the context of composite applications. It provides the foundation for building composite applications using NSF components.

We give a brief introduction to the new architecture of Lotus Notes 8 and the supported composite application components that are available out of the box. You can learn how to build NSF components based on existing Notes applications.
22.1 Composite applications in Lotus Notes 8

A major new enhancement in Lotus Notes 8 is the support for composite applications, a key element within IBM service oriented architecture (SOA) and contextual collaboration strategy. Using Lotus Domino Designer, developers can use existing Notes applications for building NSF components to be used in NSF based composite applications. A composite application can include any combination of NSF components and Eclipse components with the plan to support local running Portal components when the product will be available. Some parts of the release used for this book were not yet available.

The ability to combine multiple technologies into a single application provides significant business value: It enables companies to protect and extend their existing assets with increasing degrees of flexibility, and to respond quickly and cost effectively to their emerging business requirements, with applications that are significantly easier to assemble than alternative application development environments.

22.1.1 Terminology

Composite application is a fairly new area and not everybody is familiar with the terminology used. Here is a brief overview listing the most commonly used terms related to NSF component development in this chapter.

- Notes application
  If you are familiar with Lotus Notes and Domino this is what used to be called Notes database. You can have multiple frames, framesets, pages, forms, views as well as folders and navigators.

- NSF component
  This is basically a Notes application used in a composite application. In other words it is a composable component.

- Component view for NSF component
  This is the visual representation of a NSF component. It basically is one rectangle in the user interface and can consist of all the elements typically used in a Notes application such as frames, pages, views and others. This is also what you see when you import a NSF component view into the Composite Application Editor palette.

- NSF based application
  A composite application is described in the composite application XML file. A NSF based application describes the fact that one or more composite application XML files are stored in the NSF component.
22.1.2 Architecture overview

IBM has started to move all client side applications towards a server managed platform. The strategic framework for this is Lotus Expeditor. As foundation for Lotus Notes 8.0, Version 6.1.1 of Lotus Expeditor is used. This provides a number of advantages and allows for easy extension of existing Notes and Domino based applications with others such as Eclipse or Web based applications as well as Portlets.

Lotus Expeditor provides the foundation for the IBM server managed client platform. As shown in Figure 22-1 support for Eclipse components and local portlets are part of Lotus Expeditor already. Because the premium client of Lotus Notes 8 is based on Lotus Expeditor all features are available to you and the applications you want to deploy in Lotus Notes 8.

Among other enhancements Lotus Notes 8 add composite application support. As part of this it supports the integration of Eclipse and local portlets with NSF components.

There are also some out-of-the-box NSF components supplied by Lotus Notes 8. This includes the well known PIM features, such as e-mail and calendar as well as todo and contacts. Among many other things that are new in this release, this chapter focuses on the composite application features introduced with Lotus Notes 8.

22.1.3 Components in a composite application

Lotus Notes is based on Lotus Expeditor. It therefore supports all composite application components available for Lotus Expeditor. The supported technologies include:

![Figure 22-1 Lotus Notes 8.0 architecture](image-url)
Abstract Window Toolkit (AWT)

Embedded Browser. Web Application based user interfaces are displayed within an embedded browser view that is part of a predefined perspective provided by the platform. The embedded browser is a configurable and manageable browser that you can embed in a client application.

JSR 168 standard portlet viewer

Standard Widget Toolkit (SWT) components

Web container such as servlets and JSPs

WSRP viewer to locally display a portlet which is actually running on a remote portal server.

NSF components
Apart from the components supported through Lotus Expeditor there are additional composable components, so called NSF Components, introduced by Lotus Notes 8. NSF Components are elements of a Notes application surfaced in a composite application.

Without modifying an existing Notes application, you can use the Composite Application Editor to simply surface Notes design elements such as views, forms, documents and other elements of the Notes application as NSF component.

The following Notes design elements can be "surfaced' as components in composite applications:

- Pages
- Forms
- Views
- Folders
- Framesets
- Navigators

In order for a NSF component to interact with other components some additional elements are required as outlined in Figure 22-2. To expose information to other components through properties and act upon information from other components a WSDL file is required. It defines actions and properties for the NSF component.

To act upon the WSDL actions you need to refer those to Notes actions as well as publish Notes properties as WSDL properties as described in 22.3.7, "Creating a NSF component using “Discussion Forum.NSF”".
Lotus Notes 8 provides a number of NSF Components out-of-the-box such as Mail, Calendar, *ToDo* and Contacts. These are discussed in 22.2, “Composable components in Lotus Notes”.

### 22.1.4 Lotus Script APIs for inter-component communication

In a composite application the components are loosely coupled together. The component that enables this functionality is the property broker as shown in Figure 22-3. To enable the inter-component communication for a NSF component you need to utilize the new Lotus Script APIs related to property broker. You can find a more detailed description in “Lotus Script Library for Property Broker Functions” on page 571.
This release includes many new properties, methods, and classes in the LotusScript APIs including the following:

- **NotesSession**
  - NotesSession.GetPropertyBroker([Namespace])
    Returns NotesPropertyBroker. If no argument supplied, **Namespace** argument must be supplied to all Get/Set calls. If the **Namespace** argument is supplied, PropertyBroker is defaulted to a specific namespace. Get/Set calls can be made without the **Namespace** argument but if it is supplied, the argument overrides the default.

- **NotesPropertyBroker**
  - NotesPropertyBroker.InputPropertyContext (property)
    Returns an array of input NotesProperties. For current releases, only the first member of the array will be populated.
  - NotesPropertyBroker.Namespace (property)
    String. Provides the Read/Write default namespace of interest for PropertyBroker.
  - NotesPropertyBroker.HasProperty(Name,[Namespace])
    Boolean. Indicates whether a new property exists.
  - NotesPropertyBroker.GetProperty(Name,[Namespace])
    Returns NotesProperty.
NotesPropertyBroker.GetPropertyValue(Name,[Namespace])
Returns the value for a specified property.

NotesPropertyBroker.SetPropertyValue(Name, Value, [Namespace])
Returns NotesProperty. It creates a backend object and populates it with
the value. This method throws an error if the property name is not defined
by WSDL.

NotesPropertyBroker.ClearProperty
A new method that purges any new or modified property, not for input
properties.

NotesPropertyBroker.Publish
A new method that publishes all new and modified properties.

NotesProperty

NotesProperty.Values (property)
Read/Write for simple properties: Returns a Variant array of type String.
Single value properties contain data in first array element. As input,
accepts Variant or Array of type String. The Array must be homogeneous.
At this time, an Array can contain only one String entry. Input properties
cannot be set.

NotesProperty.Name (property)
Read only. Returns property name as string.

NotesProperty.TypeName (property)
Read only. Returns type name as string.

NotesProperty.Namespace (property)
Read only. Returns namespace as string.

NotesProperty.Title (property)
Read only. Returns title as string.

NotesProperty.isInput (property)
Read only. Returns true if this is an input property, otherwise false.
Returns Boolean.

NotesProperty.Description (property)
Read only. Returns description as string.

NotesProperty.Publish( )
A new method that publishes this property if modified.

NotesProperty.Clear( )
A new method that purges this property if new or modified.

**Note:** This Lotus Script API description is based on an early version of the M4 build of Lotus Notes 8. With the availability of the first public Beta version it is expected that there will be some changes related to the namespace functionality. Most of the namespace related methods will disappear. Check the latest documentation for any details.

### 22.2 Composable components in Lotus Notes

A NSF component is a link to a specific view, frameset, document or other design components contained in a Notes application. One Notes application can contain multiple NSF components which can be used in multiple composite applications.

To have a NSF components participate in a composite application it must provide references to the properties and actions in the WSDL.

In order to assist you in taking advantage of composite applications in your environment the Personal Information Management (PIM) applications in Lotus Notes 8 to provide some common properties and actions so that they can be consumed in composite applications.

This section describes the different NSF components which are delivered as part of the Eclipse based Lotus Notes 8.

Lotus Notes 8 has a simple yet flexible model for providing properties that can be consumed as part of composite applications. the properties that are provided are simple data types and only strings or dates are exposed. These data types are used to define properties to be exposed as part of a NSF component such as the mail or contacts components.

**Note:** The descriptions in this section are based on an early version of the M4 build of Lotus Notes 8. Most likely there will be changes related to naming and data types of certain properties and actions before the first public Beta version is available.

- **Mail**

  Each component contains properties to publish information and actions for initiating some response to a property from another component.

  The properties of the mail component are exposed in the **Inbox** folder and the **All Documents** view. The definition of these properties are stored in the
MailComponent.WSDL file. Each of the properties are assigned to certain elements such as the subject field for example.

Table 22-1 provides an overview of properties currently available for the mail component.

**Table 22-1 Properties for mail component**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Type</th>
<th>Assigned to</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmailAddressOutput</td>
<td>emailAddress</td>
<td>AvailabilityIcon (Inbox)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AvailabilityIcon (All Documents)</td>
</tr>
<tr>
<td>CommonNameOutput</td>
<td>commonName</td>
<td>Sender (Inbox)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Who (All Documents)</td>
</tr>
<tr>
<td>SubjectOutput</td>
<td>string</td>
<td>Subject (Inbox)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subject (All Documents)</td>
</tr>
<tr>
<td>NotesURLOutput</td>
<td>notesURL</td>
<td>Mapped to NotesSelectedDocument</td>
</tr>
</tbody>
</table>

The mail component provided actions are defined in the same WSDL file as the properties and listed in Table 22-2.

**Table 22-2 Actions for the mail component**

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>NewMemoUsingToField</td>
<td>This action takes any field as a parameter and places it in the SendTo field for a newly composed memo.</td>
</tr>
<tr>
<td>NewMemoUsingMailtoURL</td>
<td>This action takes a string in the format of a MailToURL, parses it, and places the contents in the appropriate fields of a newly composed memo.</td>
</tr>
<tr>
<td>NewMemoUsingEmailAddress</td>
<td>This action takes the provided e-mail address and places it in the SendTo field of a newly composed.</td>
</tr>
</tbody>
</table>

The MailTo string for the NewMemoUsingMailtoURL is based on RFC2368 which is the standard for this syntax. For more information, see also: http://wu-ftpd.org/rfc/rfc2368.html

This standard does not specify subject, CC, BCC, etc. per se, but it does allow for additional field definitions. The name/value pairs supported in this action are shown in Example 22-1.
Example 22-1  MailTo string

```
mailto == SendTo. Single value.
cc == CopyTo. Supports multiple values.
bcc == BlindCopyTo. Supports multiple values
subject == Subject of the memo
body == Body of memo
```

An example of this string would be:

```
mailto:john_doe@example.com&cc=jane_doe@example.com&cc=jimmy_dean@examplehome.co&bcc=my_boss@example.com&subject=Test
```

### Calendar

The calendar properties and actions are used to create the integration and interaction between the Calendar view and the DatePicker navigation object in the navigation pane. The properties are defined in Table 22-3.

**Table 22-3  Properties for calendar component**

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Date Parameter</td>
<td>DateType</td>
</tr>
<tr>
<td>Current Date Range Parameter</td>
<td>DataRangeType</td>
</tr>
</tbody>
</table>

The calendar component also has a number actions you can use as outlined in Table 22-4.

**Table 22-4**

<table>
<thead>
<tr>
<th>Action Name</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ChangeCalendarDate</td>
<td>Sets the date on the calendar.</td>
</tr>
<tr>
<td>ChangeNavigatorDateRange</td>
<td>Sets range of secondary visible dates in the navigation pane. These are dates that are active but not selected. For example, in a one work week view, Monday, Tuesday, Wednesday, Thursday, and Friday would be secondary visible dates.</td>
</tr>
<tr>
<td>ChangeNavigatorDate</td>
<td>Sets the selected date on the navigation pane.</td>
</tr>
</tbody>
</table>
Personal Address Book

The properties exposed in Personal Address Book are exposed in the My Contacts view only. These properties are defined in the ContactComponent.WSDL file and described in Table 22-5. Each property is assigned to element in the user interface.

Table 22-5  Properties for Personal Address Book component

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property type</th>
<th>Assigned to</th>
</tr>
</thead>
<tbody>
<tr>
<td>EmailAddressOutput</td>
<td>emailAddress</td>
<td>E-mail column</td>
</tr>
<tr>
<td>CanonicalNameOutput</td>
<td>canonicalName</td>
<td>SametimeInfo column</td>
</tr>
<tr>
<td>CommonNameOutput</td>
<td>commonName</td>
<td>Name column</td>
</tr>
<tr>
<td>StreetAddressOutput</td>
<td>string</td>
<td>Address</td>
</tr>
<tr>
<td>NotesURLOutput</td>
<td>notesURL</td>
<td>Mapped to NotesSelectedDocument</td>
</tr>
</tbody>
</table>

There are no actions defined in the Personal Address Book at this time.

22.3 Developing NSF components

Utilizing the out-of-the-box NSF components shipped with Lotus Notes 8 provides a good start. Another approach is to reuse your existing Notes applications. You can surface different elements as NSF components and integrate them with other NSF components or components supported by Lotus Expeditor as outlined in 22.1.3, “Components in a composite application” on page 549 to a new composite application. Let us have a closer look at the required steps for creating a new NSF component based on an existing Notes application.

22.3.1 Introduction

A NSF component is a link to a specific view, frameset, document, or other design component contained in a Notes application. The same Notes application
can contain multiple NSF components, and those NSF components can be used in multiple composite applications.

All components in a single Notes database share the same set of types, properties and actions defined in one or more Composite Properties design elements. To have the component participate in a composite application, it must make reference to properties and actions in the Composite Properties WSDL. A component can publish properties, or consume them, or both.

**Building NSF components**

Building NSF Components can be an iterative process and can be applied to existing Notes applications as well as to new Notes applications and includes the following steps.

1. Determine what Notes design elements you want to expose. This can be things like framesets, views or forms for example.
2. Determine the properties the component needs to publish.
3. In order to consume properties when it is wired to other components you need to determine the actions your component needs to perform.
4. The mechanisms to list the actions and properties for the component is a WSDL file that you need to create.
5. Import the WSDL you created into the Notes application where specific design elements or documents will be surfaced as NSF components.
6. Make the required modifications in the Notes application design elements to link them to the previously defined properties and actions using the new LotusScript classes, methods and properties.

**Note:** In order to use a NSF component in a portal based composite application you need to add a specific portlet representing the NSF component to your WebSphere Portal server.

**View Components**

Among the first things that you should define are the components views that you want to make available as part of a NSF component. For this use case scenario we want to have two views:

- **Customer View**
  
  This view sorts the discussion threads using a customer ID.

- **Reservation View**
  
  This view sorts the discussion threads using a reservation ID.
Figure 22-4 provides an example of a view sorted using a customer ID.

<table>
<thead>
<tr>
<th>New Main Topic</th>
<th>New Response</th>
<th>New Response to Response</th>
<th>Chat with Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CUS0001</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Missing Item in invoice (John Demo)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CUS0002</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>Wrong statement (John Demo)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 22-4  View sorted using a customer ID

**Publish and consume properties**

As mentioned, a component can publish different properties. There are two ways for a component how to do this:

1. The first option is to use the new Lotus Script APIs. You can do this anywhere Lotus Script can be executed.

2. The other way is only possible when the component is a view or folder. In this case you can associate a column to an output property defined in the WSDL. You can find this option in the Advanced tab of the column properties. Since a column contains a single value only properties with simple types can be associated with a column.

For a NSF component to consume a property from another component there is only one way. The NSF component must contain a "Notes action" that refers to a "WSDL action" which is wired to receive the property. To create an association between a Notes action and a WSDL action, use the Advanced tab of the Action Information box as shown in Figure 22-5.

Figure 22-5  Referring Notes actions to WSDL actions
When a property change event occurs, if that property has been "wired" to a WSDL action in your NSF component, and if the NSF component contains a Notes action which is associated with that WSDL action, then the Notes action is executed. Normally, the Notes action will use the new LotusScript APIs to discover what value was published, and it will do something with that value.

A Notes action is executed in response to the triggering of a WSDL action, only if the form, view or page containing that action is displayed by the component at that time.

The Hide attributes of a Notes action are not used to determine whether it will execute in response to a WSDL action. In fact, all Notes actions called by property change events, should generally be hidden at all times. Such actions will typically use the LotusScript API to read an input property from the property broker, and this would cause an error if activated by a user, since there is no input property to read at that time.

### 22.3.2 Software requirements

The following sections assume that you have successfully installed the following components:

- **Lotus Notes 8**
  - Runs the Discussion Forum application.
- **Lotus Domino Designer 8**
  - Build the NSF Component based on the Discussion Forum in the composite application. This is an option while installing the Lotus Notes 8 client.
- **Discussion Forum.NSF**
  - Available as download or you can create it based on the description in the appendix.

### 22.3.3 Existing notes application

In order to outline what needs to be done to enable your existing components we assume an existing discussion forum. As part of the car rental scenario we defined that there is an internal forum at that company which is used to support customer requests as shown in Figure 22-6.

The forum supports different views including show by customer ID or reservation ID. In the forum itself you will not find any customer or contract details. All that is stored are the different IDs. This way the forum can be opened up to personnel who is not allowed access to sensitive customer and reservation details.
The current forum has no extensions that could be used to interact with other components. Whenever the user wants to know more about a customer or a specific contract a different application is required where you can look up the customer or reservation ID.

22.3.4 NSF component design

Composite applications are aggregations of multiple components that are loosely coupled to support component reuse. Changes in data values in one component can be broadcasted through properties. These are then wired to actions in other components to receive the change notification and react to it.

Without modifying our existing Notes application you can use the Composite Application Editor to simply surface the views, forms, documents and other
elements of the Notes application as NSF components. To support
inter-component communication you have to modify the NSF component and the
elements that need to be surfaced as components.

The value of composite applications is not to replicate functionality already
available in one form or other but to combine components in order to add value.
In our case there are other applications which allow a user to look up information
about a specific customer by its ID or access contract details by using the
reservation ID.

22.3.5 Definition of Data Types, Actions, Properties and Namespace

Before you can make any modifications to your existing application you need to
think about what actions and properties your NSF component can and should
publish and consume.

When you start developing components for composite applications, teams often
come up with their own naming conventions for data types, property names,
descriptions and captions and the same for actions. While we do not have a
strong opinion as to what the right convention should be, we think it makes sense
to agree on one convention for your enterprise. This is important to make names
more readable and easier to understand. As you will learn in later chapters,
property names, action names and names of data types show up in the wiring UI
of the Composite Application Editor.

Let us have a look at conventions that we recommend you could use.

Data Types
In order for components to communicate with each other the wired elements
need to be of the same data type. So the first thing you should look at it is the
data types your component should use for actions and properties. If the type of
your field in your application corresponds directly to a general available data type
the you should use it. This way reuse of the component is easier compared to a
proprietary one.

Convention for data types:
- Data Type names should be lowerCamelCase, for example, emailAddress
- No Type in the data type name, for example, emailAddressType

In this scenario other components defined certain data types. Because the
discussion about convention started later on it was not possible to change them
in time. That means for this NSF component that you need to use the same data
types in order to be able to communicate with the existing components.
The data types that you need to define are:

- CustID
- ResID

These are both of type string, because that is the only type that is supported at the moment.

**Note:** To achieve maximal interoperability, the best practice is to use string as the type for your properties. In this case the type was given already by other components and could not be changed for a number of reasons.

**Namespace**

Composite applications bring together many data realms into one application. Two different data realms might have a type that is best described as “Category”. But the definitions of category in one realm might not make sense in another. For simplicity you want to call each of the types Category, but you want to prevent an application assembler from wiring together.

To solve this problem, there is the concept of "namespaces." Namespaces provide unique descriptors that you can use to group collections of types into. The name use for a type may exist in multiple namespaces, but they are considered unique by the composite applications editor.

Convention for namespace:

- In your company you should agree on one namespace, for example, http://com.myCompany.standardTypes

This use case scenario assumes that you integrate existing Eclipse components. These components have data types and the corresponding name space define already. Again, the discussion about conventions kicked in later and it was not possible to adjust the existing components. The name space used in these components is defined as http://www.ibm.com/wps/c2a. This means that the properties you need to expose as part of the NSF component have to use the same namespace.

**Note:** Types and namespaces are likely to be used across multiple components. Standards should be set within your company for the definitions of these items.

**Properties**

When deciding which properties you want to expose you have to decide between providing as many as possible or as much as required. There are trade-offs and decision to make. If you want to publish information you need to define output...
properties. For receiving input, you need to define input properties used by the actions.

Conventions for output properties:
- The names should be descriptive. You should use spaces.
- The names should express that some event happened, for example, changed, selected, etc.
- Good sample: Selected person’s e-mail address changed
- Bad sample: FileSave
- The programmatic names should be shorter than the display names, should not use spaces, and should be UpperCamelCase.

For this scenario we need two output properties:
- Selected Customer Changed
  Publish Customer ID of selected document
- Selected Reservation ID Changed
  Publish Reservation ID of selected document

Conventions for input properties:
- Input property names should describe the data type they have
- Good sample for input property: Person’s e-mail address
- Bad sample for input property: Selected person’s e-mail address changed

You need to define the following input properties:
- Customer ID
  Input property containing the Customer ID
- Reservation ID
  Input property contain the Reservation ID

All properties are of type CustID and ResID.

**Actions**
Actions are typically used to search and select a data element type. On the simplest level actions are the Notes definitions that can uniquely specify a document: UNIDs and Notes URLs. The next level up then are identifiers that an application defines that can also be used to identify a document. These are usually fields like the name or account ID. That is exactly what you will be using for the component here.
Conventions for actions:

- The names should be descriptive. You should use spaces.
- Actions names should describe the service they provide.
- Good sample for action: Show new memo UI using emailAddress
- Bad sample for action: UpdateUI
- The programmatic names should be UpperCamelCase.

In this scenario you need actions that can consume the properties coming from other components. Based on the these properties which happen to be also either customer ID or reservation ID the corresponding view is selected and the document with the ID highlighted as shown in Figure 22-3. This means we need two actions with an input parameter.

- **Select Document By Customer ID**
  
  Change the view to show documents sorted by Customer ID and highlight the document for this specific ID.
  
  Use SelectDocumentByCustomerID for the programmatic name.

- **Select Document By Reservation ID**
  
  Change the view to show documents sorted by Reservation ID and highlight the document for this specific ID.
  
  Use SelectDocumentByReservationID for the programmatic name.

---

**Note:** The term "action" is used in general to talk about the "plug" in a component that consumes a property value. Do not confuse the term "action" used in this sense, with the action buttons used in Notes views and forms. All components can have WSDL actions whether or not they are Notes applications.
22.3.6 Creating the WSDL file using the Property Broker Editor

A component can participate in an inter-component communication once you have specified actions and properties that can be wired to other components. The WSDL file lists those actions and properties a specific component can expose. In addition it also defines the namespace and other attributes of the various properties and actions.

Design elements of a Notes application may be exposed as multiple NSF components, but the database typically requires a single WSDL file. When Notes applications are used as NSF components, you also need to specify the properties and actions in various design elements in the database that reference the WSDL.

Using the new Property Broker Editor in Lotus Domino Designer 8 we now create the WSDL file that should reflect the data types, actions and properties as well as the name space as defined in 22.3.5, “Definition of Data Types, Actions, Properties and Namespace” on page 562.

1. Property Broker Editor can be accessed directly from Lotus Domino Designer 8 which means you need to start that first.

2. To create a new WSDL file click New Properties in the action bar. If you want to open an existing file in your notes application click Open File.

3. In the Property Broker Editor you need to create the two new data types CustID and ResID.

4. Select the Types tab and click New Type. This creates a new entry under All Types.

5. As Name type CustID, select String as the Type of your property and make sure to have http://www.ibm.com/wps/c2a as your Namespace.

6. Create a second type and define the name as ResID using the same parameters as for CustID.

7. Your types tab should look like Figure 22-8 on page 567.
8. Switch to the Properties tab in order to add the properties we defined earlier.


10. Click Add to create a new properties entry. Define the Name as Selected Reservation Changed and set the Type to ResID.

11. Make sure that for the input properties Allow programmatic publishing is not selected. Otherwise those properties will also show up in the Composite Application Editor.

12. Add the other properties using the names defined in Chapter 22.3.5, “Definition of Data Types, Actions, Properties and Namespace” on page 562.

13. Your Properties tab should look like Figure 22-9 on page 568.

**Note:** Do not use any special characters in the description for data types, actions and properties as this might produce WSDL files that contain invalid XML structures.
14. Switch to the **Actions** tab in order to create the required actions.

15. All you need is two actions that have one input parameter.

16. Click **Add** to create a new action and define the **Name** as **SelectDocumentByCustomerID**. As the property for the input parameter select Customer ID as shown in Figure 22-10 on page 569.

17. Add a second action where you define the **Name** as **SelectDocumentByReservationID** with Reservation ID as input parameter.

18. To store the WSDL file click **File → Save**.

19. At this stage you can close the Property Broker Editor.
22.3.7 Creating a NSF component using “Discussion Forum.NSF”

Now that we defined the data types, actions and properties required for the NSF Component, it is time to have a closer look at what this actually means for the existing Discussion Forum.NSF.

There are a number of extensions you need to do in order to create a NSF component that can be integrated to a composite application:

1. Import the WSDL file created in 22.3.6, “Creating the WSDL file using the Property Broker Editor”.

1. Implement some Lotus Script functions that connect the existing discussion forum functionality with the Property Broker. Here you define a couple of functions that can receive other component’s properties or publish properties from the NSF component as defined in the WSDL file.

2. Enable two views to publish a couple of properties to the Property Broker. The properties are the customer and reservation ID as defined earlier on.
3. This NSF component shall also react to properties published by other composite components. That means you need to refer the WSDL Actions to the actual Notes Actions.

4. Add a new frameset which surfaces in the NSF component and is used for assembling the composite application later on in the process.

**Import WSDL file**

In order to have support for the defined actions and properties you need to import the WSDL file you generated into the discussion forum NSF:

1. In order to work on the NSF you have to start Lotus Domino Designer 8 first.

2. To open the NSF file click **File → Application → Open** and select **Discussion Forum.NSF**.

3. In the design pane you need to expand **Composite Applications** and click **Properties**. The work pane appears.

4. To import the WSDL file click **Import WSDL** at the top of the work pane and the **Open** dialog appears.

5. Select **Discussion Forum.wsdl** and then click **Open**.

Figure 22-11 shows the imported WSDL file in the work pane.
Lotus Script Library for Property Broker Functions

In order to connect the existing discussion forum with the Property Broker we need to add a few lines of Lotus Script code. You can either inline the Lotus Script whenever it is required or define it in a Lotus Script Library and access it when appropriate. In this example we have chosen to define them once and call them. The new Script Library named PBRoutines you need to create contains a number of elements. First you need to add declarations we will use throughout the database as shown in Example 22-2 on page 572.

The declarations start with the definition of the namespace. It is important to note that the namespace for the different data types we use have to be the same as defined in the other components. That is why we agreed on using “http://ww.ibm.com/wps/ca2a”, because this was used in the majority of the other components already. When you define a composite application this is one aspect you need to take a look at in detail. Otherwise the components cannot communicate with each other.
Using the next few declarations we define constants for the different properties required to publish and receive properties in the WSDL file.

We also define constraints for accessing the text fields you can use afterwards.

**Example 22-2  Lotus Script Declarations**

```lsp
Const NAMESPACE="http://www.ibm.com/wps/c2a"
Const SET_CUSTOMERID = "Customer ID"
Const GET_CUSTOMERID = "Selected Customer Changed"
Const SET_RESERVATIONID = "Reservation ID"
Const GET_RESERVATIONID = "Selected Reservation Changed"

Const FIELD_CUSTOMERID="CustomerID"
Const FIELD_RESERVATIONID="ReservationID"
```

The next step is to implement a function named publishProperty that is used to publish a property from the database to the property broker. The implementation is outlined in Example 22-3.

It receives the required values which are property name and the property value. Please note that these are all of Type String.

The first thing we do in this function call is to define a few objects such as the current Notes Session, the Notes Property Broker as well as the Notes Property itself. Once that is done a function call is used to set the received properties and publish them to the Property Broker.

Another option to publish properties is to create a new column. You can then automatically publish this to Property Broker. This is not used here. Please see the product documentation for further details.

**Example 22-3  Implementation for publishing properties**

```lsp
Sub publishProperty(propName As String, propValue As String)
    On Error Goto propErr
    Dim s As New NotesSession
    Dim pb As NotesPropertyBroker
    Dim p As NotesProperty
    Set pb = s.GetPropertyBroker(NAMESPACE)
    Call pb.setPropertyValue(propName, propValue, NAMESPACE)
    Call pb.Publish()
theEnd:
    Exit Sub
propErr:
    Resume theEnd
```
In the same way we have an implementation for publishing properties we also need one for receiving potential action parameters from other components via the Property Broker. The code for this implementation is shown in Example 22-4.

Again we start with object declarations including the current Notes Session and Database as well as the Notes Property Broker and its Input Properties and the Property Name among some others required here.

The next step is then to use a reference to the current database in order to get access to the Property Broker for the given namespace. Using the Property Broker reference we then access the input property object and retrieve the property name and namespace.

Using the GetPropertyValue function we then access the value for this specific property. This is also the value returned by this function.

**Example 22-4  Implementation for receiving action parameters**

```vba
Public Function getActionParameter() As String
    On Error Goto theend
    Dim s As New NotesSession
    Dim db As NotesDatabase
    Dim pb As NotesPropertyBroker
    Dim pbInputProperty As NotesProperty
    Dim inputPropertyName As String
    Dim propNameSpace As String
    Dim pbValue As String
    Set db = s.CurrentDatabase
    Set pb = s.GetPropertyBroker(NAMEPASE)
    Set pbInputProperty = pb.InputPropertyContext(0)
    inputPropertyName = pbInputProperty.name
    If inputPropertyName = "" Then
        Goto theend
    End If
    propNameSpace = pbInputProperty.namespace
    pbValue = pb.GetPropertyValue(inputPropertyName, propNameSpace)(0)
    getActionParameter = pbValue
    Exit Function
theend:
    getActionParameter = ""
    Exit Function
End Function
```
These are all definitions required for the Lotus Script Library.

Note: You can reuse this Lotus Script library in other applications. The only application specific things are declarations including the namespace.

The following steps outline what you have to do in order to make those changes in the database:

1. Making changes in the Discussion Forum.NSF requires to start the Lotus Domino Designer 8 if you did not do this already.

2. Open up the Discussion Forum.NSF again and in the left pane select **Shared Code → Script Libraries**.

3. At the top of the right pane, click the **New Lotus Script Library** button, which creates a new but empty Script Library as shown in Figure 22-12.

![Figure 22-12 New Script Library](image)
4. First you need to define a name for the new library. In the right pane of the new script library, right-click and select **Script Library Properties** to open the information box for this script library.

5. In the information box you need to specify the **Title** of this new script library. Type in **PBRoutines** as shown in Figure 22-13. There is no need to add any comment to the **Comment** field. Close the information box again.

![Figure 22-13](image)

**Figure 22-13**  Title for the new script library

6. By default **Options** should be selected. Here you need to add just one statement as shown in Example 22-5. The purpose of Option Explicit is that undeclared variables will generate syntax errors.

*Example 22-5  Define Lotus Script options*

```
Option Explicit
```

7. Click **Declarations** and copy the ones we have defined in Example 22-2 on page 572.

8. Now define the function call for publishing properties. Select **Initialize** and in the right pane copy the Lotus Script from Example 22-3 on page 572 below the **End Sub** line. This creates a new function call entry in the left pane as shown in Figure 22-14.
9. Below the **End Sub** line in this function you just created copy the code from Example 22-4 on page 573 which defines the function to receive properties from other components. The Script Library should now look like the screen capture shown in Figure 22-15.
Figure 22-15 Receiving property function

10. You should not see any errors. If you do, make sure that you use the exact syntax as shown in the different examples.

11. You can now save your new Lotus Script Library by using Ctrl → s on your keyboard.
Enable Views to publish Properties
In the existing discussion forum there are two views that we want to use for publishing properties. These are “By Reservation ID” and “By Customer ID”. Whenever a user selects a document in one of those views the Customer ID and Reservation ID should be published to the Property Broker.

In the Notes database view you can access the Onselect event. It occurs when the row highlight moves to a new row within a view. In order to publish properties from a specific view we respond to this event and add an implementation as shown in Example 22-6.

Note: The implementation for responding to the Onselect event is the same for both the “By Reservation Id” as well as the “By Customer ID” view because the same two properties are published to Property Broker.

The function receives a reference to the current Notes UI View. Using that we get access to the selected document by first extracting the Note ID. With that we extract a reference to the actual document from the database.

From the document itself we then extract the property name and its property value. Using the predefined function “publishProperty” we then publish the customer ID as well as the reservation ID to Property Broker.

Example 22-6  Implementation of Onselect event to publish two properties

Sub Onselect(Source As Notesuiview)
    On Error Goto theend
    Dim s As New NotesSession
    Dim db As NotesDatabase
    Dim Doc As NotesDocument
    Dim map List As String
    Dim noteID As String

    Set db = s.CurrentDatabase
    noteID = source.CaretNoteID
    If noteID = "" Then Exit Sub

    Set doc = db.getDocumentByID(noteID)
    publishProperty GET_CUSTOMERID, doc.CustomerID(0)
    publishProperty GET_RESERVATIONID, doc.ReservationID(0)
    Exit Sub
theend:
    Exit Sub
End Sub
The following steps outline what you have to do in order to make those changes in the database.

1. You should have your Lotus Domino Designer 8 and the Discussion Forum.NSF still open. If not, open them up again.

2. In the left pane select Views and scroll down and select the By Customer ID view. The view opens in the right pane of your workbench.

3. The second half of the right pane is split into two separate panes. In the left one, select View Selection under By Customer ID (View).

4. In the right pane, replace the View Selection script with the one in Example 22-7.

Example 22-7  Modified View Selection Lotus Script

```lilac
SELECT form != "Interest Profile"& !@Contains(@LowerCase(form); "archive")
```

5. A bit further down select Options and add the definitions from Example 22-8 in the right pane. The Use “PBRoutines” statement loads the Lotus Script library you implemented earlier on. The Explicit option is used again to show errors when a variable is not defined.

Example 22-8  Define Options

```lilac
Use "PBRoutines"
Option Explicit
```

6. Scroll down in the left pane and select the Onselect event. Replace the script with the one we discussed in Example 22-6 on page 578. The result of this should look like Figure 22-16 on page 580.
7. The same changes you have to make to the By Reservation ID view by going through Steps 1 through 6.
8. Do not forget to save your changes.

**Note:** There is a limitation when you use the M4 build of Lotus Notes 8. Onselect only works for documents but not for categories in views.
Refer Notes Actions to WSDL Actions

There is only one way for a NSF component to consume a property from another component. The NSF component must contain a "Notes action" that refers to a "WSDL action" which is wired to receive the property.

When a property change event occurs, if that property has been wired to a WSDL action in your Notes component, and if the Notes component contains a Notes action which is associated with that WSDL action, then the Notes action is executed.

Normally, the Notes action will use the LotusScript APIs described earlier to discover what value was published, and it will do something with that value. In our scenario we use it as a key to select the requested view and highlight the document within that view.

We need two Notes actions for the NSF Component. One to change the view to show all documents sorted by customer ID and highlight one depending on the action value and a second one to do the same for the reservation ID view. We use the same names for the Notes actions as defined for the WSDL actions: setCustomerViewByID and setReservationViewByID.

For these two new Notes Actions you need to add a new form to the database. This form we name “DiscussionForumListActionPane” and it is a hidden one. The only task for this form is to change the views depending on the properties coming in from another component that you need to wire to these two actions.

As part of the action we implement the Click event using the code shown in Example 22-9 on page 581. It usually occurs when the user selects an action, action hotspot or button. In our scenario the referred WSDL action triggers the Notes action and therefore the Click event.

Once it occurs the first thing we do is use the Lotus Script function we created earlier to retrieve the value of action parameter. Then we set the “DiscussionForumListClient” Frame as current database so that we can open the “By Customer ID” view within that frame. Using the value of the action parameter we select the corresponding document within that view.

Example 22-9 Implementation for Click Event for setCustomerViewByID

```
Sub Click(Source As Button)
    Dim s As New NotesSession
    Dim newDoc As NotesDocument
    Dim workspace As New NotesUIWorkspace
    Dim uiDb As NotesUIDatabase
    Dim newName As String
    Dim value As Variant
```

newName = getActionParameter()
If newName = "" Then Exit Sub
Set uiDb = workspace.CurrentDatabase
workspace.SetTargetFrame "DiscussionForumListClient"
uiDb.openView "By Customer ID", newName, False, True
End Sub

Almost the same Lotus Script code can be used for the second Notes Action that
we refer to the second WSDL Action. This one is responsible for the “By
Reservation ID” view the Notes Action that selects this view. In Example 22-10
you can see the implementation for the Click event.

Example 22-10   Implementation for Click Event for setReservationViewByID

Sub Click(Source As Button)
Dim s As New NotesSession
Dim newDoc As NotesDocument
Dim workspace As New NotesUIWorkspace
Dim uiDb As NotesUIDatabase
Dim newName As String
Dim value As Variant

newName = getActionParameter()
If newName = "" Then Exit Sub
Set uiDb = workspace.CurrentDatabase
workspace.SetTargetFrame "DiscussionForumListClient"
uiDb.openView "By Reservation ID", newName, False, True
End Sub

The first step is to create a new form which is called
“DiscussionForumListActionPane”. Within that pane you then need to add two
new Notes actions. To do that, follow the steps outlined below.

1. Assuming that you are still in the Lotus Domino Designer 8 and the
   Discussion Forum.NSF is still open, click Forms in the left pane. This creates
   a new form.

2. In the top right pane move your mouse to the right edge until the curser
   changes into a slider. Hold down the left mouse button and move the slider to
   the left. The upper pane in the right window should now be split and your
   screen should look like Figure 22-17.
3. Right-click in the new pane that you just opened. In the menu that opens up, click **Create Action** which opens up an information box.

4. In the Name field type in **setCustomerViewByID**.

5. Since this function should be hidden and only used when a WSDL Action is wired to this Notes Action you have to deselect **Include action in Action bar** as well as **Include action in Action menu**.

6. In the programmer’s pane at the bottom right there are two drop-down lists at the top of that pane. Use the second one and select **LotusScript** so that we can add the script as discussed earlier.

7. First you need to make sure to add the **PBRoutines** and the **Explicit** option again by copying the code from Example 22-11 into the **Options** section.

---

**Example 22-11** Options definition for setCustomerViewByID

Use "PBRoutines"
Option Explicit
8. Now add the implementation for the Click event from Example 22-9 on page 581.

9. Repeat Steps 3 to 8 for the setReservationViewByID using the implementation from Example 22-10 on page 582.

10. In order to save your newly created Form, press Ctrl → s on your keyboard. This brings up a Save Form as dialog. Type in **DiscussionForumListActionPane** and select OK.

Once you have completed the Form it should show the two actions as in Figure 22-18.

*Figure 22-18  Completed form with two hidden actions*
This was the first part. The second part is to refer the two Notes Actions you just created to the WSDL Actions.

11. Select Notes Action **setCustomerViewByID** and open up the information box for that action. right-click and select **Action Properties**.

12. Go to the Advanced tab which is the one at the right end.

13. At the bottom of that pane there is a section **Composite Settings** and a drop-down list. From that list select **SelectDocumentByCustomerID** as shown in Figure 22-19. This is the corresponding WSDL Action and you just refered the Notes Action to the WSDL Action.

14. Do not forget to save your form again.

![Figure 22-19  Referring Notes Action to WSDL Action](image)

15. Repeat Steps 11 through 13 for the Notes Action setReservationViewByID using the corresponding WSDL Action SelectDocumentByReservationID.

**Creating the NSFComponent**

As described a NSF Component can surface different Notes design elements. This includes Pages, Forms, Views, Folders as well as Framesets or Navigators.

In our composite application we use a frameset to surface as NSF Component. It shows a view depending on the corresponding actions. So for example if another component is using the defined action for selecting and sorting the database by customer ID the corresponding view is selected and a document highlighted that contains the customer ID as defined in the property received from the other component.

For this we create a new frameset with two columns. The first one is hidden and contains the new form “DiscussionForumListActionPanel” which you created already. This form is responsible for switching the view in the second column which is using either the “By Customer ID” or “By Reservation ID” view depending on the wired action.
The following steps outline what you have to do in order to make those changes in the database.

1. You should still have your database open in Lotus Domino Designer 8.
2. In the left pane click **Framesets**. This opens up the list of existing framesets on the right.
3. You need to create a new frameset. In the action bar click **New Frameset**. This opens up the **Create New Frameset** dialog as shown in Figure 22-20.
4. Use the default selections and click **OK**.

![Create New Frameset dialog](image)

*Figure 22-20  Create New Frameset dialog*

5. You have now created a split frameset with two columns but no content so far. Right-click in the left column and click **Frame Properties** which opens up the information box.
6. In the **Name** field enter **DiscussionForumActions**.
7. As **Type** you need to define **Named Element** using the first drop-down list. In the one that shows up, next to it select **Form**.
8. To specify the Form you created earlier on use the **Browse** button next to the **Value** field.
9. This brings up the Locate Object dialog in which you just use the default values for **Kind of object** and **Application**. At the bottom use the drop-down list to select the DiscussionForumListActionPane as shown in Figure 22-21 on page 587 and click **OK**.
10. Close the information box for the left column of the frameset.

11. Right-click in the right column and click **Frame Properties** to open up the information box for this column.

12. In the Name field type DiscussionForumListClient.

13. Define **Named Element** as the **Type** and select **View** from the drop-down list that appears next to it.

14. Click Browse button next to the **Value** field to open the **Locate Object** dialog again.

15. Using the default settings you can use the drop-down list at the bottom to select **By Customer ID view** and click **OK**.

16. The information box for the second column should look like Figure 22-22 on page 588 which you can close now.

---

**Figure 22-21** Locate the form used in the frameset
17. Your new frameset has now two columns but only the right one is showing something and should look like Figure 22-23 on page 589.

18. Since we do not want the first one to be visible for the user, move your mouse to the line that divides the two column until the cursor changes to a slider.

19. Holding down the left mouse button drag the slider towards the very left end of the frameset so that nothing is visible of the first column.
20. In the action bar click **Save Frameset** to save all your settings.

21. A dialog appears. Type **DiscussionForumList** as new name for this frameset and click **OK**.

22. You are done.

### 22.4 Tester development

One of the things that you need to consider when you develop NSF components is a way to actually test them. There are multiple ways of doing this. One of the ways is to develop a tester component.

#### 22.4.1 Overview

A tester component should be capable of dealing with all actions and properties available in your composite component. What tooling you use to develop such a
tester component is not relevant as long as you have a mechanism to deploy them in the same environment as your component that you want to test.

Actions and properties
In this scenario we implement the tester component as a NSF component in the same Notes application. Typically you would want to have it as part of a separate Notes application.

- Publish Customer ID
  Publish Customer ID entered in field
- Publish Reservation ID
  Publish Reservation ID entered in field

There are also some actions available in the discussion forum that are exposed in the NSF component. The tester has to be able to access those. In this case they are:

You need to define the following input properties:

- Customer ID
  Input property containing the Customer ID
- Reservation ID
  Input property containing the Reservation ID

You realize that in this case we have a symmetric set of properties and actions. That means you can implement the same actions and properties in the tester that you have in the NSF component representing the discussion forum. That is not always the case.

Actions for Tester Component:

- Display Customer ID
  Displays the Customer ID in a text field.
  Use DisplayCustomerID for the programmatic name.
- Display Reservation ID
  Displays the Customer ID in a text field
  Use DisplayReservationID for the programmatic name.

If you do not have a symmetric set of properties and actions you need to add additional functionality in your tester component to make sure you can handle all required options.
**User interface**

A tester should be simple and easy to handle. Typically you want to have one line for properties and potential corresponding actions.

As shown in Figure 22-24 you need to implement two lines in a table. First you have the Customer ID related properties and actions. It consists of a brief description of context of the property and action, a text field that can be used for both input and output and a button to trigger an action.

The text field displays the value of a property received by the tester component. In a symmetric setting you can use it also as input parameter for the corresponding action which is triggered by the button.

The second line provides the same functionality for the Reservation ID.

![Figure 22-24  Tester view elements](image)

**22.4.2 Tester implementation**

The implementation of this tester is rather simple. You can reuse the code introduced for creating the NSF component for the discussion forum.

**Visual elements**

The following steps provide some guideline how to create the visual elements of the tester.

1. Using Lotus Domino Designer 8 you need to open the Notes application created in Chapter 22.3.7, “Creating a NSF component using “Discussion Forum.NSF” on page 569.
2. Create a new Notes application and name it TesterComponents.nsf.
3. Add a new Form to the Notes application.
4. In that form create a table with two rows and three column.
5. In the first column add the context of your properties and actions such as Customer ID and Reservation ID.
6. In the second column add simple text fields to handle the customer and reservation ID.
7. In the last column add buttons that you can latter on use to publish the text you enter in the fields as property to Property Broker.
8. Below the table add a computed text field, define its name as SaveOptions and set the Value to "0". This prevents Lotus Notes from asking if you want to change the changes when you close it.

9. The visual components should look similar to Figure 22-25.

![Figure 22-25 Visual elements of tester component](image)

**Programmatic elements**

Similar to what you did for the discussion forum you also need to publish properties and respond to the WSDL actions. Here a brief description what you need to do. You can use the same code and Lotus Script library developed in Chapter 22.3.7, “Creating a NSF component using “Discussion Forum.NSF” on page 569.

1. Create a new WSDL file defining the same data types as for the discussion forum and add the actions and properties as outlined above.

2. To publish the properties you use the Click method of the buttons as shown in Figure 22-26. You can modify and reuse the code you have developed earlier. It is documented in Example 22-3 on page 572.

3. You need to add two Notes actions that you can refer to the WSDL actions. In the Notes actions you need to implement the code for receiving action parameters responding to the Click event. You can use and modify the code you have developed earlier. It is documented in Example 22-4 on page 573.

4. Update the Declarations to match the properties defined for the Tester component as shown in Example 22-12.

**Example 22-12  Lotus Script Declarations**

```livescript
Const NAMESPACE="http://www.ibm.com/wps/c2a"
Const SET_CUSTOMERID = "Customer ID"
Const GET_CUSTOMERID = "Publish Customer ID"
Const SET_RESERVATIONID = "Reservation ID"
Const GET_RESERVATIONID = "Publish Reservation ID"

Const FIELD_CUSTOMERID="CustomerID"
Const FIELD_RESERVATIONID="ReservationID"
```
5. Make sure to refer the Notes actions to the WSDL actions using the information box.

![Tester development in Lotus Notes Designer 8](image)

**Figure 22-26 Tester development in Lotus Notes Designer 8**

**Note:** If you do not want to develop this component you can use the provided Notes application which contains the Tester component.

### 22.5 Lotus Domino server deployment

Now that you do have the NSF components developed you can use them locally or deploy them to the Lotus Domino server. For testing purposes it is alright to have them locally. If you want to use the NSF component in a composite application that you want to role out to other users as well you need to deploy your application on the server.
For the following chapters we assume that the discussion forum has been deployed to a Lotus Domino server. For this purpose a Lotus Domino Server 7 was used.

There are multiple ways of deploying a Notes application to a Lotus Domino server. How you do it depends on your company policies, your admin rights and many other things. The steps below just describe one way of doing it. That is making a copy to the server.

1. Start your Lotus Notes 8 client.

2. To open the Notes application click File → Open → Lotus Notes Application.

3. A file dialog appears where you need to locate the DiscussionForum.NSF file. It should be stored in the view but if you stored it in a different directory you need locate it there. Select Discussion Forum and click Open to open the Notes application.

4. Once open the next step is to create a copy on the server. Click File → Application → New Copy to the Copy Application dialog.

5. Specify the Server where you want to place your copy which is stdom/itso.

6. The application Title and the File name should be filled out already as shown in the capture in Figure 22-27 on page 595.

7. At the bottom you can specify what should be copied. Make sure that both Application design and documents as well as Access Control List are selected.

8. To start the copy process, click OK.

9. Once finished a message in the status bar at the bottom should say Database Copy Completed. This means you have deployed your application to the Domino Server so that other users can access it from there.
10. Publish the TesterComponent Notes Application as well by repeating these steps.

You are ready to move on and assemble a composite applications.
Utilities for assembling composite applications

In general, you can divide the development of composite applications into two phases. Phase one deals with individual component development that uses the development tools appropriate for that person and environment. Typically software engineers are responsible for this phase. The second phase is more related to business processes and a different set of people that usually do not have software development skills. Because of this, the utilities for assembling composite applications have a different target group. They are aimed toward the business person.

This chapter provides an overview of available utilities for assembling composite applications. The following utilities are available:

- Composite Application Editor
  This is part of Lotus Notes 8 and provides support for portal and NSF based composite applications. The Composite Application Editor is described in this chapter.

- WebSphere Portal tools
  This is part of WebSphere Portal with support for portal based composite applications. The WebSphere Portal tools are described in Chapter 3, “Utilities for developing composite applications” on page 51.
23.1 Composite Application Editor

The Composite Application Editor allows you to edit active applications that reside on an IBM Lotus Domino server or on IBM WebSphere Portal. You can also edit application templates that reside on WebSphere Portal.

The changes that you make are reflected immediately in the application. You can also use templates to build new applications. The changes that you make to a Portal application template apply to applications you create from the template after saving the changes. You can also save changes to a new Portal template.

Before you can use the Composite Application Editor (CAE) you have to make sure it is installed on your system. The CAE comes with the Lotus Notes 8 client. When you install the client you need to make sure to select the check box as shown in Figure 23-1.

If the CAE is already installed on your system, you can open the application or template and choose Actions → Edit Application.

![Figure 23-1 Select the Composite Application Editor during install]

23.1.1 The user interface

To assemble or edit an existing NSF based composite application you just have to open it in Lotus Notes. For a portal based composite application you want to access templates and applications that reside on a portal server. In this case you need to make sure that the Home Portal Account in Lotus Notes 8 is set up correctly. More details about this can be found in Chapter 24, “Assembling composite applications using the Composite Application Editor” on page 605.
The user interface is more or less the same, independent of whether you work against a Lotus Domino or WebSphere Portal server. When you are in editing mode, using **Actions → Edit Application** of a newly created NSF based composite application, the user interface (UI) of the CAE consists of three main areas as shown in Figure 23-2.

- On the left-hand side you can see an application menu pane which lists all components that are part of your application. This provides you with direct access to edit application elements. In this example it just shows a blank page.
- The center of the workspace displays the application components for the current page. Again, that is an empty page when you start from a new composite application template.
- On the right you find a component palette which lists components you can add to your application. The components shown here might differ somewhat from what you are seeing as part of your screen. You do not have to worry about that now as we make sure that you have all components there when you need them.

![](image)

**Figure 23-2** User Interface of the Composite Application Editor in editing mode

### 23.1.2 Assembling composite applications

In this chapter we give a general overview about the most important elements you must use to assemble composite applications. How to assemble a specific
application is described in Chapter 25, “Lotus Notes 8 sample scenario” on page 673.

**Component palette**
When you assemble a new composite application the first thing you want to make sure is that you have all components and add them to the component palette. The following components you can add to your palette:

- NSF components
- Eclipse Components from an update site
- Eclipse Components from your local machine

To add components to the components palette, right-click in the component palette view and select **Add Components → NSF Component** for NSF components, for example. There are two other menu items referring to Eclipse components either from an update site or your local machine. Depending on what component you want to import you either have to point to an update site or the directory where your components are stored.

By default when you import components which are no PIM components they will show up under General. It is up to you however how you want to organize this and you can add other palettes if you want to.

When you work against a portal server the options for import components change slightly. In this case you build a portal based composite application instead of a NSF based composite application. The composite application XML is then stored and deployed via your WebSphere Portal server instead of using a Notes application and Lotus Domino server. In this mode the following options are available for importing components:

- Add a Portal component
- Create a new Portal component
- Import Eclipse component from an update site
- Import Eclipse component from your local machine

Using the Create a new Portal component option you can create rich client portlets for non-portal components such as NSF or Eclipse components. This creates a place holder portlet that you can use to assemble your composite application either using the Composite Application Editor or the WebSphere Portal tools. If an application that contains these placeholder portlets gets deployed it accesses the Eclipse or NSF components from its original location to instantiate it in the client. So, for example, when you have created a new Portal component based on a Eclipse component stored in an update site, the components get installed directly from the update site on the rich client.
Adding a component to your application

Once you have the components in the component palette you can just drag and drop them using the right mouse button from the palette to the center of your workbench. This will add the component to the application.

While you are in drag and drop mode the cursor changes. When you add the first component you will see just a folder symbol. Starting with the second one you will either see a small arrow pointing either left or right, up or down. That means you can position the component next, above or below the existing component. If a folder symbol appears, it will create a new tab and place the component in the same view as the existing one. Just try it out and place your components in different places. Figure 23-3 shows an example for this.

![Composite Application Editor in editing mode](image)

**Figure 23-3  Composite Application Editor in editing mode**

Note: When adding additional components to your Component Palette make sure that the user of this application has access to the components as well. Only references to these components are stored in the Composite Application XML created by the Composite Application Editor.
Any component you have added will show up on the left-hand side as an application component. From there you can edit its properties directly.

**Creating wires**
Assuming you have all components positioned in your application, you can use the CAE to create wires between the components. Using these wires, the components can then exchange information and trigger certain actions other components can listen to.

The nice thing about creating wires using the CAE is that you do not have to write any code.

In order to get to the wiring screen just select one of the application components listed on the left-hand side of the workbench and right-click. In the menu that appears, select **Wiring** and a new screen appears. This screen contains all the components that you have placed in your application. It also shows any available properties you have access to as start or end points for creating a wire.

By default, the component you had selected when opening the wire view is the one that selected a source component. All other components are potential end points for creating a wire.

To create a wire, select a property from the source component, left-click and keep the button pressed. Drag the mouse pointer across to the other component you want to wire up and an arrow will appear. Since only properties of the same type can be connected, the CAE will refuse the ones that do not match. If it is a compatible data type, then just drag the arrow to the property and release the mouse button. The wire has been created. Figure 23-4 is a good example of wires using the Composite Application Editor.

![Composite Application Editor in wiring mode](image)
Finished Editing
Once you are done with editing your application all you need to do is close the CAE. To do this, just use File → Finished Editing which closes the editing mode. Note that the application is not updated when you leave the editing mode. It is actually updated whenever you make some changes to it.
Assembling composite applications using the Composite Application Editor

Composite applications are aggregations of multiple components that are loosely coupled. Changes in data values in one component can be broadcast through properties. These are then wired to actions in other components to receive the change notification and to react to it.

There are multiple options for assembling composite applications. In this chapter you can learn how to use the Composite Application Editor to create and edit active applications that reside on an IBM Lotus Domino server and IBM WebSphere Portal.
24.1 Introduction

Composite applications are a key element in a service-oriented architecture (SOA) and contextual collaboration strategy. The ability to create and edit composite applications lets you easily integrate different types of components and technologies.

Server managed composite applications can be grouped into two categories:

- NSF based composite applications
  - Based on Notes applications. They can consist of NSF, Eclipse and Portal components. The composite application XML created using the Composite Application Editor is based in a new design element part of a Notes application.

- Portal based composite applications
  - The composite application XML is stored and defined on your portal server using either the Composite Application Editor or the WebSphere Portal tools. They can also consist of NSF, Eclipse and Portal components.

By creating composite applications, you can loosely couple NSF, Eclipse and Portal components in context of a specific business need and create a new user experience. Components within a composite application can be wired together so a user-initiated action in one component can trigger a programmed action in another component.

You can define composite applications using the Composite Application Editor in the Notes rich client or the WebSphere Portal Application Template Editor in the browser, both of which are described in this section. If you want to host your composite application on Lotus Domino you need to use the Composite Application Editor to define the composite application. If you host them on WebSphere Portal you can use either of the two editors mentioned.

24.1.1 NSF based composite applications

A composite application based on a NSF file works similar to what you have read so far about the composite applications assembled using Portal. They share the same Composite Application XML which defines the application, but there are also some differences.

Overview

A composite application can be stored in a NSF file either locally on your computer or hosted on a Domino server. To automatically launch a composite
application when you open an NSF file in the Notes Standard client, you must store the composite application in the Notes database.

A NSF based composite application is defined through a .xml file. This .xml file is stored in a new design element in the NSF file.

**Note:** The Composite Application XML which defines a NSF based composite application is identical with the one used for a Portal based composite applications.

That means you define your application by defining what components you want to use. When you open a NSF file in Lotus Domino Designer 8 you should see the Composite Application design element as shown in Figure 24-1. In Properties you stored the WSDL file as outlined in the previous chapter. Into the Application design element you can either import an existing .xml file of an existing composite application or create a blank one. Using the Composite Application Editor you can then create a new composite application based on your existing NSF file.

![Composite Applications](image)

*Figure 24-1 Composite Application design element*

If you create a new composite application, you can create the NSF application using the supplied Blank Composite Application template using Lotus Notes 8. The Composite Application design element behaves just like other design elements in terms of replication, template inheritance, and so on. You may have more than one of them in a single Notes database, though this is unusual.

In addition to storing the Composite Application XML in the Notes application, you must also change certain settings to tell the Notes client to launch the Composite Application when the user opens the Notes application. If you use the Composite Application template for creating a new NSF file this has already been done for you.

**Composite Application Structure**

You have two options for your NSF based component. It is possible to extend an existing Notes application by adding the Composite Application XML structure to the NSF file or to create a new NSF file based on the Composite Application template. In either case the composite application is defined in the XML file.
You can have Notes or Eclipse components integrated to one application. While assembling your composite application you should be aware of a few basic things regarding the structure and how a composite application based on a NSF works.

Let us assume you build a new composite application as outlined in “Creating a new composite application” on page 609. You just have one Notes Component and a Eclipse Component. First, make sure that both components are accessible by the user of your composite application. Typically, the Notes Component and the corresponding NSF file are stored on a Lotus Domino Server. The user can access it there directly or replicate it down to the client.

The Eclipse component is provided via an update site. Update sites are explained in Chapter 15, “Exporting components and publishing the update site” on page 375 in more detail. It can either be located on your machine or on a HTTP server hosting the update site.

In the case when you create your new composite application an additional NSF file is created. During development time you most likely have it on your development machine. Later on it is deployed also on a Lotus Domino Server for example.

As shown in Figure 24-2 on page 609 you have three files that you have to manage in this scenario. How the actual deployment of your composite application works is discussed in Chapter 15, “Exporting components and publishing the update site” on page 375. However, during development you need to think about where your composite application components are located.

Why is that you might think? The reason is because all you do when you assemble your composite application using the Composite Application Editor for example is to create the Composite Application XML file. This describes the instructions to create the composite application on the client. It is the clients responsibility then to look up all the defined components and aggregate them into one graphical representation. That means however that when defining the XML structure you need to make sure that components are accessible by the user later on.

In our little example you have to make sure that the Lotus Domino Server and the Notes Component are accessible. At the same time the Eclipse Component which needs to be integrated has to be available on the update site somewhere in the network. This could be on the users machine as well but in reality it is easier to manage when you have it on a HTTP server.
Chapter 24. Assembling composite applications using the Composite Application Editor

The NSF that contains the composite application can be either deployed on the Lotus Domino Server like any other NSF based application or locally on the client's computer.

**Creating a new composite application**

The typical steps for creating a new NSF based composite application are more or less always the same. A much more detailed description is provided in 24.1.1, “NSF based composite applications” on page 606 which follows the steps outlined here:

1. Create a new NSF file using Lotus Notes 8 and the new Composite Application Template.
2. Open the new application in Lotus Notes 8 and start the Composite Application Editor.
3. Import all required components and position them accordingly.
4. Create the wires to allow for inter-component communication.
5. Deploy the application using Lotus Domino Server.
Extending an existing NSF application

When you do have an existing Notes application you can turn this into a NSF based composite application following these steps:

1. You need to open the existing Notes application in Lotus Domino Designer 8.
2. Utilizing the new design elements for composite applications you need to create a new Composite Application XML file in the NSF application.
3. In addition you must also change certain settings to tell the Notes client to launch the composite application when the user opens the Notes database.
4. Once you have done this you can open your Notes application in Lotus Notes 8 and define the NSF based composite application using the Composite Application Editor.
5. Deployment is just the same way as before using Lotus Domino Server.

24.1.2 Portal based composite applications

The Portal based composite application approach was explained in detail in earlier chapters. The same techniques and concepts that apply to Lotus Expeditor do also apply for Lotus Notes 8. That means that you can deploy and manage the same applications you might have for Lotus Expeditor today also in Lotus Notes 8.

One of the additional features that you get when creating a portal based composite application is the role based provisioning. That means you can define certain roles and assign your application to those different roles. This way you can easily decide who should see what application.

In addition you can build composite applications that consist of all the composite application components supported by Lotus Expeditor and integrate them with Notes Components. That is true for both the NSF and Portal based composite applications.

For Portal based applications you need to perform a few additional steps to make a NSF component available to Portal. This is described in detail in 24.5, “Assembling a Portal based composite application” on page 643.

1. Develop your NSF component as outlined in one of the previous chapters.
2. Create portlets that represent the NSF component by creating a unique instances of the Domino Application Portlet also called DAP.
3. Deploy the portlets which represent the NSF components on WebSphere Portal.
4. Create your portal composite application using the Portal tools or the Composite Application Editor.
24.2 Assembling a NSF based composite application

Assembling a composite application is the easy part once you have all components in place. Let us walk through the steps required in order to create a new NSF based composite application and assemble different components.

24.2.1 Introduction

The NSF based composite application discussed in this section consist of three different components. It contains one Eclipse and two NSF components that are assembled and wired. The user interface that contains these three components is captured in Figure 24-3 on page 612.

The idea is to have a discussion data base which so far was used as a mechanism to discuss certain issues regarding car or van rentals. Because of the sensitivity of customer and rental details a decision was made to only use the IDs and no customer names or store reservation details. The problem with this is that you need other applications to obtain more about the customer details. Combining the discussion forum with other applications dramatically improves the efficiency of the personnel who needs to understand the details behind the customer ID.
The case study scenario implements the functionality that when a user selects a document in the NSF view component at the bottom the customer ID gets published. The other two components listen to that event as shown in Figure 24-4 on page 613 which represents wiring used in the Composite Application Editor.
In the case of the Eclipse view component on the top left it looks up a local database in order to display the data associated with the customer ID. The second NSF view component on the top right just displays the customer ID as well as the reservation ID. In addition you can type in either customer or reservation ID which get published. Depending on the ID the NSF view component changes to the selected view and highlights the corresponding document in the discussion forum. If it is a customer ID, the Eclipse view component looks up the details again.

### 24.2.2 Software required

In order to create this new composite application you need to make sure the following software and components are available to you:

- **Lotus Notes 8**
  The Eclipse based or standard client as it is called is required for this.

- **Composite Application Editor**
  During install of Lotus Notes 8 you need to select the Composite Application Editor as shown in Figure 24-5 on page 614

- **Eclipse Component**
  In Chapter 19, “The ITSO Car Rental sample composite application” on page 477 it is explained how to build the different Eclipse components. To complete this section you need access to the customer information and the database component as well as the Cloudscape database. These are available to you on the update site.
24.2.3 Composable components

Let us have a closer look at the components you need to use.

- NSF components
  There are two NSF components. The NSF views represent a form the other one a frameset. They are stored in two different Notes applications.

- Eclipse components
  The Eclipse component that we use consist of multiple features. First there is the customer details feature and implements the user interface related aspects. In addition it establishes a connection to a local database using the database feature. The database feature uses a local database which in this case is Cloudscape.

  Be default Lotus Notes 8 does not ship with a local database because it is not required. That is why Cloudscape is also available as a feature on the update site that contains the Eclipse component. When you use the Eclipse component for the first time it automatically installs all other features that are required, including Cloudscape.
Before you start to assemble components as part of your composite application you need to think about where you place these components. Be aware that the location needs to be the same and accessible for the user of your application.

Figure 24-6 on page 615 outlines the different locations of the components. The Eclipse component and all required features are stored in an update site on the HTTP server that is part of IBM Domino Server. The NSF components are stored on a Lotus Domino server.

![Figure 24-6 Location of composable components](image)

For testing purposes you can also have them stored on your computer. In that case however nobody else will be able to use the application unless they have the files stored in the same location in the file system.

### 24.2.4 Creating a new composite application

A composite application is defined in the composite application XML file. When creating a NSF based composite application this XML is stored in a Notes application. When creating new composite application you create a new file for this. When turning an existing Notes application into a NSF based composite application you can choose between creating a new Notes application that contains the composite application XML file or to extend your existing Notes application. In this scenario we create a new one.

During the testing and development phase you want to have your new composite application locally. Once you are done you can deploy this to the server. The scenario here assumes that all components you assemble are already deployed to a server but the new composite application is created locally. Afterwards you need to deploy it to the Lotus Domino server.
1. In order to create a new NSF based application you need to start the standard (not the basic) version Lotus Notes 8. That is the new client based on Eclipse. Only this version supports composite applications.

2. Make sure you are have the Home tab or another Notes application selected. Only then the menu items are visible you need to use.

3. Select File → Application → New to open the New Application dialog. Here you need to specify the details of your new application.

4. Make sure to have Local selected from the Server drop-down list.

5. Type in the Title you want to give your new application such as ITSO Composite Application.

6. The File name should be ITSO Composite Application.nsf and completed automatically by default.

7. In the bottom half of the dialog you can select an application template. The Server drop-down list should show Local as the selected location here as well for you to see the local templates.

8. From the Template list select -Blank Composite Application-. Compare your settings with Figure 24-7.

9. If they are the same click OK in order to create the new application.
10. Lotus Notes 8 created the new application and should have opened it after that automatically.

11. Figure 24-8 on page 618 shows a capture of the newly created application.
24.2.5 Assembling the components

Now that you have an empty NSF based composite application let us look at the integration and assembling of the different components. There are basically three steps that you have to complete:

1. Import the Eclipse and NSF components into the Composite Application Editor palette.
2. From there you can place the Eclipse and NSF views for the components in your layout.
3. Once that is complete use the wire feature available in the Composite Application Editor to establish the inter-component communication.

As outlined in 24.2.3, “Composable components” the components that you need to assemble are available on the Lotus Domino and HTTP server.

Note: In order to deploy your application to users make sure to use components that are available to the users as well.
Importing components
In order to use the components you need to import them into the Composite Application Editor.

1. Make sure Lotus Notes 8 is started and the blank composite application is open.

2. To open the Composite Application Editor click **Actions → Edit Application** and a new window opens.

3. Click the right arrow on the right border in order to open the Component Palette as shown in the screen capture of Figure 24-9.

4. To import the components create a new category by right-clicking in the Component Palette area. In the menu click **Add Category** and an entry field opens at the bottom of the palette. Type in **ITSO Components** and press **Enter**.

5. A new category appears in the palette.

6. In order to start importing components right-click the new category just created.

7. In the menu that appears, click **Add Components → Add NSF Component** which brings up a dialog.

8. In the **Component Name** field type **Discussion Forum View**.
9. In the Component Description field type View component of discussion forum.

10. To locate the NSF component click Browse next to the Notes URL field. This brings up a Locate Object dialog.

11. Click the icon which looks like a folder next to the Application drop-down list to open the Choose Application dialog.

12. The components that you need to import are located on the server. That is why you need to select stdom/itso in the Look in drop-down list.

13. From the list below that should be visible now select ITSO Discussion Forum.

14. Click Select to close this dialog.

15. Back in the Locate Object dialog define the Kind of object as Frameset.

16. At the bottom choose DiscussionForumList and your dialog should look like Figure 24-10. This represents one component view of your NSF component.

17. Click OK to close this dialog.

![Figure 24-10 Locate frameset in discussion forum application](image)

18. Back in the previous dialog check that ITSO Components is selected in the Category drop-down list.

19. The Notes URL might be different but the other entries should be the same as in Figure 24-11 on page 621.

20. Click OK to close this dialog.
21. You just added a composite component to your palette.

22. Repeat steps 6., “In order to start importing components right-click the new category just created.” on page 619 through to 21., “You just added a composite component to your palette.” on page 621 for the second component view of your NSF component using the TesterComponents.NSF application.

23. In this case Component Name should be Tester View. In the Component Description field type in View component of Tester form.

24. In the Locate Object dialog make sure to select the same Notes application as before. In the Kind of object drop-down list you need to select Form. At the bottom select Tester in the Form drop-down list.

25. Once you have completed these steps a second NSF component should appear in your ITSO Components palette.

You have added two NSF components to the palette in the Composite Application Editor. Next you need to add the Eclipse component from the update site which is also located on the Lotus Domino Server.

1. Right-click the ITSO Components palette and click Add Components → Add Eclipse View from Update Site.

2. This opens a dialog where you need select Remote Update Site.

3. You need to know the URL where your update site is located. In this scenario, use http://9.42.170.221:8088/updatesite/site.xml.

4. The Update Site Details dialog should look like Figure 24-12.
5. Click **OK** to close it.

![Update Site Details](image)

*Figure 24-12  Remote update site located on Lotus Domino Server*

6. A progress bar appears while the update site is being parsed.

7. Once finished a dialog box appears. In the Components view select **ITSO customer details**. There is a second item available from that update site named **ITSO synchronize** and adds synchronization capabilities to the application. Select that component as well. You have the option to add this later on in the process.

8. At the bottom make sure that **ITSO Category** is selected as shown in Figure 24-13.
9. Click **OK** to close this dialog.

10. You have imported four components to the palette which should look like Figure 24-14 on page 624.
Now that all components required for this composite application are available in the Component Palette you can start working on the visual integration.

There are three components, two NSF and one Eclipse components. For this a simple layout can be used. The two Eclipse view components do not require as much space on the screen compared to the NSF component. That is why we place the NSF view component at the bottom of the screen and the two Eclipse views next to each other above the NSF view as shown in Figure 24-15 on page 625.
Figure 24-15  Composite application layout

The following steps are required to create this layout using the three view components.

1. You should have the Composite Application Editor and the blank composite application still open.
2. Open the Component Palette on the right side of the editor again.
3. From the ITSO Components palette select Discussion Forum View.
4. Use the mouse to drag and drop this view component to the blank page in the middle of the editor. Once placed there, a progress bar appears that determines the status of deploying the NSF component.

**Note:** When you drag and drop a NSF view component into a page you place the view but at the same time the Notes application is accessed and opened up as well.

5. Once successfully deployed you should see a view similar to Figure 24-16 on page 626.
6. What you also can see is that in the left pane a element was added to the page structure. It is named Discussion Forum View which is the name you assigned to it while importing the component to the component palette.
7. Go back to the ITSO Components palette and select the Tester View.

8. Using your mouse, drag and drop the component into the page as well. This time however you need to make sure you place it in above the other component. You do that by move the cursor up to the top of the page until you see an error point upwards.

9. The editor view should look like Figure 24-17 on page 627.

10. A second element now appears in the page structure in the left pane. It is named Tester View which you defined earlier, during importing the component.
11. Positioning Eclipse view components works the same way as NSF view components. There are however differences during deployment.

From the **ITSO Components** palette select the ITSO customer details component.

12. Drag and drop it into the page and place it in the top left corner. A dotted frame should show up above the discussion forum view and next to the tester view on the left. In addition a little arrow pointing left has to be there. If that is the case, just drop the view.

13. This triggers are series of installs you have to go through. Essentially what happens is that the Eclipse component you just placed in the page is deployed from the update site into Lotus Notes 8. Meaning that the Eclipse feature is installed. A screen as in Figure 24-18 on page 628 appears. It provides details such as plug-in name as well as identifier and file identifier.

14. At the bottom you need to select **Install this plug-in** which triggers the actual install.
15. This Eclipse component depends on other Eclipse feature in order to work properly. Unless you have these features installed already two more plug-ins need to be installed.

16. The first one is the **ITSO Car Rental Database plug-in** as shown in Figure 24-19.

---

**Figure 24-18  Install ITSO Customer Details plug-in**

**Figure 24-19  Install ITSO Car Rental Database plug-in**
17. The second one is the actual database component used to store the data. In this case it is **Cloudscape** as shown in Figure 24-20, an open source database.

![Cloudscape in IBM Lotus Notes](image)

*Figure 24-20  Install Cloudscape plug-in*

18. In the page structure you should have a third element named **ITSO Customer Details**.

19. If you close the left side bar you should see all the three view components sitting next to each other, similar to Figure 24-21 on page 630.
Inter-component communication
In some scenarios a visual integration might be enough. Most likely however you want the different components to interact with each other as well. That is why you defined actions and properties for each of the components that can be connected.

In this use case scenario there are basically two different types of information to be exchanged, that is, customer ID and reservation ID. The two NSF components have properties to expose this information and actions to act upon them. The Eclipse component has just a action to consume a customer ID.

In Figure 24-22 on page 631 you can see the basic wiring you need to do between the components. From both NSF components you have to wire the properties for customer ID to the corresponding action of the Eclipse component indicated by the arrows pointing from the NSF components to the Eclipse component.

The NSF components can deal with both the customer and the reservation ID. Both also have actions and properties you need to wire. In the Composite
Application Editor for exchanging the customer ID both ways means that you have to create two wires.

Figure 24-22  Component wiring

Follow these steps to create the inter-component communication for the three components.

1. Your Composite Application Editor should still have the composite application open.

2. Open the left side bar if you closed it and right-click Tester View. In the menu click **Wiring** to open the wiring view.

3. You should see three boxes with Tester View defined as **Source** like in Figure 24-23 on page 632. You can also see that when you place your cursor over one of the actions or properties more details like type or description are displayed.
4. What you want to do is to create a wire from the two Tester View properties to the corresponding actions of the other two components. This way you can type in a customer or reservation ID and they get send to the other components.

Select the **Publish Customer ID** of the **Tester View** component and have a look at the actions of the other two components. You should notice that depending on your property selection the compatible actions are visible marked with a small circle in front of the action name as shown in Figure 24-24 for the **Discussion Forum View**.

To create the connection between the Tester and Discussion Forum View hold down the left mouse button on **Publish Customer ID** and drag it towards **Select Document By Customer ID** in the **Discussion Forum View**. An arrow should appear. Once the your curser is over the action let it go and a wire between the two components should appear as in Figure 24-25 on page 633.
5. Repeat this using **Publish** Reservation ID and connect it with the Select Document By Reservation ID.

6. In the same way you can create a wire between the Tester View and the **ITSO Customer Details** component. This one can only handle the customer ID. So use the **Publish** Customer ID from the Tester View and connect it with the **CustomerID Action**.

   As you can see the icons in front of the actions and properties change to reflect that there is a wire attached to them. You can delete a wire at any time by clicking the x-shape icon next to the action name.

7. Now change the source component to be the **Discussion Forum View**. Right-click in the title bar of the component and click **Select as wire source**.

8. Follow the same instructions as explained in the previous steps to create the same wires. The names of the actions and properties should be the same.

   Your wiring view should look like Figure 24-26 on page 634 when you click one of the properties in the Discussion Forum View.
9. To close the wiring view click **OK** at the bottom right of the page. This brings you back to the previous screen.

**Edit properties**
Before you close the Composite Application Editor you might want to update some of the application, page or component properties. The ones you need to update are:

- Application title and description
- Page title and description
- Add the current page to the frequently-used applications
- Define a default page when the application starts

The next few steps explain how to make these modifications.

1. In the Composite Application Editor click **Edit → Edit Application Properties** to open the Application Properties dialog.

2. The **Title** of the application should be **NSF Composite Application** which you need to type in the title field.

3. Below that type a description like **NSF based composite application utilizing NSF and Eclipse components** in the **Description** field. Figure 24-27 reflects these changes.

4. Click **OK** at the bottom of the dialog to close the dialog and save the changes.
5. To change the page properties right-click **Blank Page → Edit Page Properties** in the left side bar.

6. In the dialog that opens up define the **Page Title** as **NSF Composite Application**.

7. As **Page Description** type in **Main page of NSF based composite application**.

8. Click **OK** at the bottom of the dialog to close the dialog and save the changes.

9. You are done with editing the application and can click **Done** in the upper section of the left sidebar to close the Composite Application Editor.
10. Once the Composite Application Editor is closed your application appears in Lotus Notes 8 again and is ready to be deployed.

### 24.3 Composite application deployment

You have assembled a new NSF based composite application which is stored locally right now. Since we want to give other users access to this application as well you need to deploy it to the Lotus Domino server. All components that you have assembled are available on the server already.

#### 24.3.1 Deployment on Lotus Domino server

The following steps outline what you have to do in order to deploy your application on a Lotus Domino Server. This is a somewhat simplified scenario because in a corporate environment certain company policies will apply.
1. Start your Lotus Notes 8 client.

2. Open your workspace view by clicking **Open → Applications → Workspace** and locate the application icon representing the ITSO Composite Application as shown in Figure 24-29.

![ITSO Composite Application on Local](image)

*Figure 24-29  Application icon*

3. Right-click the icon and click **Application → New Copy** to open the **Copy Application** dialog.

4. Specify the **Server** where you want to place your copy which is `stdom/itso`.

5. The application **Title** and the **File name** should be filled out already as shown in the capture in Figure 24-30.

6. At the bottom you can specify what should be copied. Make sure that both **Application design and documents** as well as **Access Control List** is selected.

![Copy Application "ITSO Discussion Forum" dialog](image)

*Figure 24-30  Copy Application to Lotus Domino Server*

7. To start the copy process select **OK**.
8. Once finished a message in the status bar at the bottom should say **Database Copy Completed**. This means you have deployed your application to the Domino Server so that other user can access it from there.

### 24.3.2 Provisioning to Lotus Notes 8

Using Lotus Notes 8 you can access the composite application like any other Notes application. Just point your client to the Domino server and locate the Notes application.

The difference between a composite and Notes application is that when Eclipse components are part of the application they are getting installed when you open the composite application the first time.

At this point you need accept the installation of the Eclipse plug-ins in order for your application to work properly.

The composite application design elements of your Notes application behave like any other design element in terms of replication for example.

### 24.4 Rich client support for WebSphere Portal

Using a standard WebSphere Portal install provides support for assembling and deployment of traditional portal based applications. For supporting rich clients such as Lotus Expeditor as well as Lotus Notes 8 additional features are required which are:

- **Network Client Installer**
  
  Enhances a HTTP server to allow a remote installation of the Lotus Expeditor client to a desktop or mobile computer. In addition it extends WebSphere Portal with support for Eclipse components.

- **Lotus Notes Server Install**
  
  It includes the Eclipse component support from the Network Client Installer and extends it with NSF component support for WebSphere Portal as shown in Figure 24-31.
For a portal based composite application that you want to deploy to Lotus Notes 8 you need to make sure to install Lotus Notes Server Install on your portal first as described in 24.4.1, “Enable WebSphere Portal for Lotus Notes 8” on page 639.

Once your portal has rich client support you need to prepare Lotus Notes 8 to be able to interact with your WebSphere Portal install. This is explained in 24.4.2, “Connect Lotus Notes 8 to WebSphere Portal” on page 641.

### 24.4.1 Enable WebSphere Portal for Lotus Notes 8

The out-of-the-box installation of WebSphere Portal cannot be used to support rich clients. Additional software is required. In order to support Lotus Notes 8 clients you need to install Lotus Network Server Install V8.0.

1. Make sure your WebSphere Portal server is up and running before you start installing the Lotus Network Server Install V8.0.

2. Locate the directory for this component and start `setupwin32.exe` when you are on a Windows server. There is a version available for Linux as well.

3. A message box pops up for you to select the language you want to use for the wizard. Select **English** and click **OK**.

4. A welcome screen appears that explains that you are about to install Lotus Network Server Install V8.0 on your WebSphere Portal. Click **Next** to continue.

5. You see a license agreement you should read and accept. Select **I accept both the IBM and the non-IBM terms** and click **Next**.

6. On the next screen you need to specify the directory for this install. You might want to use the default directory which is `C:\Program Files\IBM\Lotus\Notes\Server`. Click **Next** to continue.
7. The next screen requires to specify the install directory of WebSphere Portal. The field should already say `C:\Portal\PortalServer` for this install as shown in Figure 24-32.

8. You also need to enter the **User ID** which is `wpsadmin` as well as the **Password** which is `passw0rd`. Once completed click **Next** to continue.

![WebSphere Portal server information](image)

**Figure 24-32  WebSphere Portal information**

9. After you clicked on Next the installer checks the connection to portal. A screen appears that says **Verifying connectivity to Portal**.

10. If you have entered the correct information and your portal server is up and running this verification should complete successfully. As a result, a summary screen appears that confirms the installation location, the feature you are about to install, and the required disk space as shown in Figure 24-33 on page 641.

11. If this information is correct click **Install** and the installation process starts.
12. Once the install is complete a screen will show up confirming this. It also states that you need to restart your portal server. Click OK.

13. Another screen appears, where you click Finish and you are done.


### 24.4.2 Connect Lotus Notes 8 to WebSphere Portal

Before you can create a Portal based composite application you need to connect your Lotus Notes 8 client to the WebSphere Portal server. The first thing that you need to do is install the Lotus Notes Server Client Install as explained in the previous section.

**Note:** In order to allow communication using Port 80 on portal you need to make sure the HTTP server is configured appropriately. The following steps assume communication between the rich client and portal using Port 80.

Once that is complete you need to set up the so called Home Portal Account. This is used to connect your Lotus Notes 8 client to the home portal server. All that you can change here is the description of the account and the details for your portal server. You cannot delete this account. The following steps outline what you need to change.

1. Start the standard version of Lotus Notes 8.
2. Open the Preferences page by clicking File → Preferences.
3. In the Preferences page click Home Portal Account. The account details show up on the right.

---

**Figure 24-33** Installer summary

IBM Lotus Notes Server install will be installed in the following location:

- C:\Program Files\IBM\Lotus\Notes\Server

with the following features:

- Portal Server

for a total size:

- 57.1 MB
4. In the **Server** field type in the address of your portal. In this scenario it is **http://9.42.170.223/wps**

5. In the **Name** field you need to type in the portal administrator name. In this scenario, it is **wpsadmin**.

6. The corresponding password is **passw0rd** and you need to type this into the **Password** field.

7. The **Home Portal** and **Authentication** URL should be completed automatically as the same as in Figure 24-34.

8. The **Authentication Type** should be set to **J2EE-FORM**.

![Home Portal Account settings](image-url)

9. Once you have entered all the details click **OK** at the bottom of the page to close and save the details.

10. A message box might pop-up saying **Continuing will update your portal account configuration. Do you wish to continue?** and you click **Yes**.

11. To make sure that you do have access to your portal server click **Open → Portal Applications** and you should see all available applications that you can use, similar to Figure 24-35 on page 643.
24.5 Assembling a Portal based composite application

With Lotus Notes 8 you have the choice of deploying your composite application through a Lotus Domino server or to use WebSphere Portal. In 24.4, “Rich client support for WebSphere Portal” on page 638 you created a composite application which can be deployed using Lotus Domino. This section outlines how you can use the same technique for assembling the same components but for a portal based composite application using a WebSphere Portal server.

For the portal based composite application you can use the same scenario as described for the NSF based composite application in 24.2, “Assembling a NSF based composite application”. Figure 24-36 shows the NSF based composite application. The portal based composite application looks almost the same, but it is created and assembled and deployed using the portal server.
24.5.1 Introduction

As outlined the scenario is the same as for the NSF based composite application. The components to be assembled consist of two different types of components:

- NSF Components
  Consists of the discussion forum and the tester component.

- Eclipse Component
  Consists of the customer details component that views the details for a specific customer ID.
24.5.2 Software required

In order to create this new composite application you need to make sure the following software and components are available to you:

- **Lotus Notes 8**
  The Eclipse based or standard client as it is called is required for this.

- **Composite Application Editor**
  During install of Lotus Notes 8 you need to select the Composite Application Editor.

- **WebSphere Portal 6.0.1**
  This is required because the composite application is hosted and deployed through WebSphere Portal. Make sure you know the exact URL as well as the user ID and password to access your Portal installation. That is required to allow Lotus Notes 8 access to the Portal.

- **Lotus Notes Server Install 8**
  This is a new component available with the Lotus Notes 8 client offering which needs to be installed on top of WebSphere Portal. If you are familiar with the Network Client Installer of Lotus Expeditor, this is the corresponding component available for Lotus Notes 8. It uses the Network Client Installer component and extends it so that a Lotus Notes 8 client and NSF components are supported in a composite application environment. How to add the rich client support to WebSphere Portal can be found in 24.4, “Rich client support for WebSphere Portal” on page 638.

- **Eclipse Component**
  Chapter 19, “The ITSO Car Rental sample composite application” on page 477 explains how to build the required customer details portlet. Make sure this component is available to you in the update site on the HTTP server. Additional information about the Eclipse component can be found in 24.5.3, “Composable components” on page 645.

- **NSF Component**
  The discussion forum and tester component developed in Chapter 22, “NSF component development” on page 547 has to be either on your local machine or on a Lotus Domino Server. Additional information about the NSF component can be found in 24.5.3, “Composable components” on page 645.

24.5.3 Composable components

Let us have a closer look at the components you need to use.

- **NSF component**
There is the discussion forum you used in the previous scenario. The file that you need to use is ITSO Discussion Forum.nsf. The TesterComponents.NSF contains the Tester application that you need to assemble as well. If you want to create an application that can be deployed to other users you need to use the server copy as documented in Chapter 22, “NSF component development” on page 547. The Lotus Domino server used in this scenario is on IP 9.42.170.221 as shown in Figure 24-37. For testing purposes you can also use a local copy of the Notes application. The assembling process would be the same.

**Eclipse component**

It consists of multiple features that are installed together once you use the Eclipse component for the first time.

- **ITSO Customer Details**
  
  Implements the user interface related features to display the required customer information that it is extracting using the ITSO Car Rental Database feature.

  The package that you need to have in the update site is com.ibm.itso.compappcarrental.customerdetails.feature_1.0.0.

- **ITSO Car Rental Database**
  
  Implements the required features for creating and accessing the local database which stored the customer details. It is using the Cloudscape database.

  The package that you need to have in the update site is com.ibm.itso.compappcarrental.database.feature_1.0.0.

- **Cloudscape**
  
  An open source database which by default is not part of the Lotus Notes 8 client install. That is why you need to add it to the update site. It should be available to you either via Lotus Expeditor 6.1.x or provided as part of the Lotus Notes 8 install CD. At the time of writing this was not entirely clear.

  In any case the package that you need to have in the update site is org.apache.derby.feature_10.1.3.1-20061108.

  The way the Eclipse component is build it will install all required features if they are not installed already. If some of them are only the ones that are not will be installed.
24.5.4 Creating a new composite application template

In order to assemble a new portal based composite application you need to create a new template in portal first. Lotus Notes 8 provides an application that lets you access, create and modify templates in WebSphere Portal directly. You can access it clicking **Open → Portal Templates** which opens the template catalog as shown in Figure 24-38.

The template catalog consists of two panes. The left pane shows the different categories such as Composite Applications or **My Portal**. The right pane lists all available templates for these specific categories.

Figure 24-37 Location of composable components

![Figure 24-37 Location of composable components](image)

Figure 24-38 Portal templates available to use

![Figure 24-38 Portal templates available to use](image)
You have two options for creating a new template for your portal based composite application:

- **Copy template**
  
  Select one of the existing templates and click **More Actions → Copy** in order to create an exact copy the existing template.

- **Create new template**
  
  You can create a new template using one of the existing templates as starting point. Click **New** in the action bar and a dialog appears. In that dialog you need to specify the name of your template and the starting point.

The following steps take you through the creation of a new template.

1. In the Portal Templates view, click **New** in the action bar.
2. This opens a dialog in which you need to specify the **Name** of your template as **Notes CA Template**.
3. Select **Composite Application** as **Category** for this new template.
4. At the bottom you need to choose a **Starting Point**. In this case select **Expeditor MyClient Sample Template**. This is a rich client template that gets installed via the Lotus Notes Server Install.
5. Once your selections look the same as Figure 24-39, click **OK**, which creates the new template for you.
Your new template should now show up in the Notes Template view along with the other ones.

24.5.5 Assembling the components

Now that you have created the template in the portal server you can create your portal based composite application based on that. To do that select the template you created, for example Notes CA Template and click Edit in the action bar. This initiates the creation of a local copy of the template. Based on this template the Composite Application Editor can work on.

Let us look at the integration and assembling of the different components. There are basically three steps that you have to complete:

1. Import the Eclipse and NSF components into the Composite Application Editor palette by creating placeholder portlets that can be deployed through WebSphere Portal.

2. From there you can place the portlets representing the Eclipse and NSF views for the components in your layout.

3. Once that is complete use the wire feature available in the Composite Application Editor to establish the inter-component communication.
As outlined in 24.2.3, “Composable components” the components that you need to assemble are available on the Lotus Domino and HTTP server.

**Note:** In order to deploy your application to users make sure to use components that are available to the users as well.

**Importing components**

In order to be able to use the NSF and Eclipse components to need to import them first. When you open a portal template using the Composite Application Editor it shows a slightly different behavior when it comes to importing components to the palette. You have the following options when you right-click **Add Components** in the component pallet:

- Add Portal Component
- Create New Portal Component
- Add Eclipse View from Update Site
- Add locally installed Component

Before you can import components you need to open the template you created.

1. In the **Notes Template** view select the **Notes CA Template** and click **Edit** in the action bar.

2. A **Edit Template** message pops up as shown in Figure 24-40. Click **OK** to confirm this.

3. This opens the Composite Application Editor and the new template.

![Edit Template](image)

*Figure 24-40  Edit Notes CA Template*
In order to use non portal components in a portal based composite application you can create a new portal component based on your NSF or Eclipse component. Doing this you add a new portlet in portal that you can also use with the portal tools. In this scenario you need to add the two NSF components as well as the Eclipse component as new Portal components as outlined in the following steps.

1. Open the right side bar containing the components palette and right-click **Add Components**.

2. This opens a dialog that lets you select what component you want to add a portlet for. As shown in Figure 24-41 you need to select NSF Component and click **OK**.

![Figure 24-41 Type of component to be created in portal](image)

3. In the New NSF Component dialog you need to specify the **Component Name** as **Portlet - Tester View**. We add Portlet to mark the component as a portlet which is not really required. Just makes it easier to keep the overview.

4. Enter a **Component Description** such as **Portlet representation of Tester NSF component**.

5. Click **Browse** to open the **Locate Object** dialog.

6. Click the folder icon and locate the **TesterComponents** on the Lotus Domino server **stdom/itso**.

7. Select **Form** from the **Kind of object** drop-down list.

8. At the bottom select **Tester** from the **Form** drop-down list.

9. If your details look the same as in Figure 24-42 you can click **OK**.

10. You have now defined the location of the NSF component. Make sure the Notes URL starts with notes:/// as shown in Figure 24-42.
11. Select **ITSO Components** from the Category at the bottom so that your components get placed in there.

12. Click **OK** to close this dialog and add this component to the component palette.

![New NSF Component](image)

Figure 24-42  Details for NSF component

13. Repeat the last few steps for the second NSF component using **Portlet - Discussion Forum View** as **Component Name** and the **ITSO Discussion Forum**.

14. As **Component Description** use **Portlet representation of Discussion Forum component**

15. The third component that you need to add is the Eclipse component. Here you could use the ITSO customer details component in the palette that you might have added earlier. As an alternative you can create a portlet out of the Eclipse component as well which then creates a portlet in the portal server.

16. For creating a portlet from the Eclipse component right-click in the component palette and click **Add Components**

17. In the dialog that opens up select **Eclipse View from Updatesite** and click **OK**.

**Note:** In the M4 build of Lotus Notes 8 the Notes URL sometimes starts with Notes:///.... If this is the case change the capital N of Notes into a regular n as in notes. Otherwise creating of the portlet on the portal server might fail.
18. The **Update Site Details** dialog box appears in which you need to select **Remote Update Site**, because you want to use the same update site as before which is stored on the HTTP server of Domino.

19. Type in the URL as `http://9.42.170.221:8088/updatesite/site.xml` as shown in Figure 24-43 and click **OK**.

![Update Site Details](image)

*Figure 24-43  Remote update site located on Domino server*

20. The Composite Application Editor analyzes the update site and brings up the **Select Component** dialog as shown in Figure 24-44 on page 654.

21. Select the **ITSO customer details** entry in the **Components** list.

22. At the bottom use the drop-down list to select **ITSO Components** as the **Category** where this new component should be placed and click **OK**.

23. You should have three new components in the component palette.
In this scenario no portlets are used. You can however add existing portlets available in the portal server quite easy to your component palette. If you open the Add Components dialog and select Add Portal Component the Composite Application Editor looks up all available portlets that you can use.

A dialog similar to Figure 24-45 on page 655 appears. In this, you can actually see the components you added earlier, such as the ITSO customer details or the Portlet - Discussion Forum.
Once you have imported all components added to the Component Palette you can start working on the visual integration.

We use the same scenario as in Chapter 24.2.5, “Assembling the components” on page 618. There are three components, one NSF and two Eclipse components. For this a simple layout can be used. The two Eclipse view components do not require as much space on the screen compared to the NSF component. That is why we place the NSF view component at the bottom of the screen and the two Eclipse views next to each other above the NSF view as shown in Figure 24-46 on page 656.
The following steps are required to create this layout using the three view components.

1. You should have the Composite Application Editor and the composite application template application still open.

2. Open the Component Palette on the right side of the editor again.

3. From the ITSO Components palette select Portlet - Discussion Forum View.

4. Use the mouse to drag and drop this view component to the blank page in the middle of the editor. Once placed there a progress bar appears that determines the status of deploying the component.

5. Once successfully deployed, you should see a view similar to Figure 24-47 on page 657.

6. What you also can see is that in the left pane a element was added to the page structure. It is named Portlet - Discussion Forum View which is the name you assigned to it while importing the component to the component palette.

Figure 24-46 Composite application layout
7. Go back to the ITSO Components palette and select the Portlet - Tester View.

8. Using your mouse, drag and drop the component into the page as well. This time however you need to make sure you place it in above the other component. You do that by move the cursor up to the top of the page until you see an error point upwards.

9. The editor view should look like Figure 24-48 on page 658.

10. A second element now appears in the page structure in the left pane. It is named Portlet - Tester View which you defined earlier, during importing the component.
11. Positioning Eclipse view components works the same way as NSF view components. There are however differences during deployment.

   From the **ITSO Components** palette select the ITSO customer details component.

12. Drag and drop it into the page and place it in the top left corner. A dotted frame should show up above the discussion forum view and next to the tester view on the left. In addition a little arrow pointing left has to be there. If that is the case, just drop the view.

13. If you have not deployed the Eclipse components before, this triggers a series of installs you have to go through. Essentially what happens is that the Eclipse component you just placed in the page is deployed from the update site into Lotus Notes 8. Meaning that the Eclipse feature is installed. A screen as in Figure 24-49 on page 659 appears. It provides details such as plug-in name as well as identifier and file identifier.

14. At the bottom you need to select **Install this plug-in** which triggers the actual install.
15. This Eclipse component depends on other Eclipse feature in order to work properly. Unless you have these features installed already two more plug-ins need to be installed.

16. The first one is the **ITSO Car Rental Database plug-in** as shown in Figure 24-50.
17. The second one is the actual database component used to store the data. In this case it is **Cloudscape** as shown in Figure 24-51, an open source database.

![Cloudscape in IBM Lotus Notes](image)

**Figure 24-51  Install Cloudscape plug-in**

18. In the page structure you should have a third element named **ITSO Customer Details**.

19. If you close the left side bar you should see all the three view components sitting next to each other, similar to Figure 24-52 on page 661.
Inter-component communication

In this use case scenario there are basically two different types of information to be exchanged, that is, customer ID and reservation ID. The two NSF components have properties to expose this information and actions to act upon them. The Eclipse component has just an action to consume a customer ID.

In Figure 24-53 on page 662 you can see the basic wiring you need to do between the components. From both NSF components you have to wire the properties for customer ID to the corresponding action of the Eclipse component indicated by the arrows pointing from the NSF components to the Eclipse component.

The NSF components can deal with both the customer and the reservation ID. Both also have actions and properties you need to wire. In the Composite Application Editor for exchanging the customer ID both ways means that you have to create two wires.
In order to wire the different components you need to follow these steps:

1. Your Composite Application Editor should still have the composite application open.

2. Open the left side bar if you closed it and right-click Portlet - Tester View. In the menu click Wiring to open the wiring view.

3. You should see three boxes with Portlet - Tester View defined as Source like in Figure 24-54 on page 663. You can also see that when you place your cursor over one of the actions or properties more details like type or description are displayed.

Note: The steps required to create the wires for these components are the same for the NSF based composite application.
4. What you want to do is to create a wire from the two Tester View properties to the corresponding actions of the other two components. This way you can type in a customer or reservation ID and they get send to the other components.

Select **Publish** Customer ID of the **Portlet - Tester View** component and have a look at the actions of the other two components. You should notice that depending on your property selection the compatible actions are visible marked with a small circle in front of the action name as shown in Figure 24-55 for the **Portlet - Discussion Forum View**.

To create the connection between the Portlet - Tester and Portlet - Discussion Forum View, hold down the left mouse button on the **Publish Customer ID** and drag it towards the Select Document By Customer ID in the **Portlet - Discussion Forum View**. An arrow should appear. Once the your curser is over the action let it go and a wire between the two components should appear as in Figure 24-56 on page 664.
5. Repeat this using the **Publish** Reservation ID and connect it with the Select Document By Reservation ID.

6. In the same way you can create a wire between the **Portlet** - Tester View and the **ITSO Customer Details** component. This one can only handle the customer ID. So use **Publish Reservation ID** from the Portlet - Tester View and connect it with the **CustomerID Action**.

   As you can see the icons in front of the actions and properties change to reflect that there is a wire attached to them. You can delete a wire at any time by clicking the x-shape icon next to the action name.

7. Now change the source component to be the **Portlet** - Discussion Forum View. Right-click in the title bar of the component and click **Select as wire source**.

8. Follow the same instructions as explained in the previous steps to create the same wires.

   Your wiring view should look like Figure 24-57 on page 665 when you click one of the properties in the Discussion Forum View.

---

*Figure 24-56  Wire between Tester and Discussion Forum View*
9. To close the wiring view click **OK** at the bottom right of the page. This brings you back to the previous screen.

**Edit properties and define roles**

Before you close the template you might want to define certain properties. One of the advantages of creating a portal based composite application is the concept of roles.

- **Edit Template Roles**

  By default you have to different roles defined in WebSphere Portal. This is the administrator and user role. You can get to the template roles by clicking **Edit → Edit Template Roles** which opens the Roles page of portal as shown in Figure 24-58 on page 666.

  The default settings are that your template can by an administrator only. Users with an administrator role can edit your template and manage members. Users can just use the application but do not make any changes.
In the Composite Application Editor you can also modify the template properties by clicking **Edit → Edit Template Properties**.

This opens the Template Properties settings as shown in Figure 24-59 on page 667. You can modify things like the title of the template, add a description and change the category for the template.
24.5.6 Create a portal based composite application

1. To access existing or create new portal applications, click **Open → Portal Applications**, which presents different categories in the left pane and the corresponding applications of that category in the right pane, similar to Figure 24-60.
2. Click **New** in the action bar to open the **Create Application** dialog.
3. Type in **ITSO Notes** Composite Application as the **Name** or your application.
4. Add a **Description** such as **Composite application based on** Lotus Notes in the field below as shown in Figure 24-61.
5. Once complete click **Next** at the bottom of the page.

![Create Application](image)

**Figure 24-61  Create Application details**

6. Select **Choose a template from the template library** in the dialog that appears and click Next to continue.
7. From the list of different categories choose **applications**.
8. From the templates select **Notes CA Template** that you created previously as shown in Figure 24-62 on page 669.
9. At this point you can click Finish and your new application is generated in the portal and should show up in the list of applications.
24.5.7 Composite application deployment

In order to deploy this composite application to a user there are a few things you need to do.

You need to create and deploy your composite application on your WebSphere Portal server. In addition make sure to assign roles and users to the application.

The user on his end needs to set up his Lotus Notes 8 client so that he can access the WebSphere Portal server as outlined in Chapter 24.4.2, “Connect Lotus Notes 8 to WebSphere Portal” on page 641.

Once complete your composite application will show in the Portal Application view of their Lotus Notes 8 clients as shown in Figure 24-63 on page 670.
In the Portal Applications view all the applications assigned to this specific owner. To open an application just double-click the one you need. This triggers the provisioning process and an instance of your composite application is created on the client.

During the provisioning operation you might see a dialog similar to Figure 24-64. In this figure the Eclipse components are being installed from the update site.

You will be asked if you want to install the Eclipse plug-ins. At this point you need to confirm this so that the application can work.

The NSF components are opened like any other Notes application stored on your Lotus Domino server.
Once the provisioning process is complete the application starts up and is ready for use. Using the Portal Application view you can also remove this application again.
Lotus Notes 8 sample scenario

This chapter describes a Lotus Notes 8 sample scenario. One of the rich clients used in this use case scenario is Lotus Notes 8 Beta. This is the first version that brings in support for composite applications.
25.1 Composable components

You will create a composite application based on three different components. Two NSF components and one Eclipse component.

- NSF component
  
  There is the discussion forum that you need to extend and turn into a NSF component. In addition you build a Tester component that is integrated with the discussion forum.

- Eclipse component
  
  It implements the user interface to view customer details. It consists of multiple features that are installed together once you use the Eclipse component for the first time.

25.2 Composite application types

As part of this use case scenario you need to assemble and create a composite application. There are basically two different types of composite applications that you need to create using the same application scenario:

- NSF based composite application
  
  This type of composite applications holds the composite application XML file in a Notes application. You can either extend an existing application or create a new application.

- Portal based composite application
  
  The composite application XML is stored in your portal server. From there it gets deployed and an instance created in you rich client.

Lotus Notes 8 provides support for both types.

25.3 Assembling the composite application

Using utilities such as the Composite Application Editor you need to assemble the three components to a composite application. The user interface looks pretty much the same for both the NSF and portal based composite application as shown in Figure 25-1 on page 675.
Chapter 25. Lotus Notes 8 sample scenario

The idea is to have a discussion database which so far was used as a mechanism to discuss certain issues regarding car or van rentals. Because of the sensitivity of customer and rental details a decision was made to only use the IDs and no customer names or store reservation details. The problem with this is that you need other applications to obtain more about the customer details. Combining the discussion forum with other applications dramatically improves the efficiency of the personnel who needs to understand the details behind the customer ID.

The case study scenario implements the functionality that when a user selects a document in the NSF component view at the bottom the customer ID gets published. The other two components listen to that event as shown in Figure 25-2 on page 676 which represents wiring used in the Composite Application Editor.
In the case of the Eclipse component view, on the top left it looks up a local database in order to display the data associated with the customer ID. The second NSF component view on the top right just displays the customer ID as well as the reservation ID. In addition you can type in either customer or reservation ID which get published. Depending on the ID the NSF view component changes to the selected view and highlights the corresponding document in the discussion forum. If it is a customer ID the Eclipse view component looks up the details again.

25.4 Application and component deployment

The different components that you need to assemble are deployed to a number of different servers as shown in Figure 25-3. The NSF components are deployed to a Lotus Domino Server. The Eclipse component is deployed on a HTTP server using the one provided by the Domino server.

The NSF based composite application you need to deploy to the Lotus Domino server is from where it is accessible to other Lotus Notes 8 users.

The portal based composite application gets deployed on portal. The deployment of the application to other users takes place through the portal as well.
Figure 25-3  Location of composable components
Provisioning the Lotus Expeditor clients with Device Management

This chapter describes the architecture and configuration steps that are necessary to manage Lotus Expeditor Clients using the Device Management facilities provided by Lotus Expeditor Server.
26.1 Overview

The increasing number of mobile devices on the market and the growing complexity of the software in these devices are the biggest challenges to the adoption of mobile and wireless data services.

Device Manager offered by Lotus Expeditor provides a flexible and robust system for the management of a wide variety of pervasive devices. These devices include personal digital assistants (PDAs), handheld PCs, PCs, subnotebooks, cellular phones, set-top boxes, in-vehicle information systems, and other devices for pervasive computing. Some common device groups (or device classes) are PocketPC (Windows CE) devices, Open Mobile Alliance Device Management (OMA DM), and OSGi-enabled devices, including Windows 32 devices and Linux devices.

Device Manager supports open standards management protocols, such as the Open Mobile Alliance Device Management (OMA DM) and the Open Service Gateway Initiative (OSGi).

Device Manager is used to enroll devices into a database and perform many tasks related to the management of those devices, such as:

- Device Configuration - Setting device parameters for hardware or software
- Inventory Collection - Collecting software or hardware information from the device
- Software Distribution - Distributing, installing, and removing software and data files for the device
- Initial provisioning - Providing initial access for the devices to the Device Manager server and restoring the original Device Manager configuration

Device Manager is built on a Web application server model. It consists of a set of J2EE servlets running on WebSphere Application Server. The device management data storage is in a relational database, such as DB2. Device Manager also requires an agent on the managed device which is used by the device to interact with the server.

26.2 OSGi overview

OSGi (Open Service Gateway Initiative) specifications define a standardized execution environment for services connected over a network. Provisioning an OSGi service platform to a networked device provides the ability to manage software in those devices. Examples of such devices are automotive devices (telematics), smart home devices (home appliances, security systems, and
energy management systems), and mobile devices (cellular phones and PDAs). Software can be installed, updated, or removed without having to interrupt the normal operation of the device.

26.3 OMA DM overview

OMA DM is a specification created by the Open Mobile Alliance (OMA) organization for management of wireless devices. It is a standardization that allows Device Manager to write one protocol engine to encode and decode the messages passed between the Device Manager server and the OMA DM device agent. Device manager provides an OMA DM Management Server that is a certified OMA DM 1.1.2 and OMA DM 1.2 implementation.

Note: The OMA DM protocol was formerly known as the SyncML/DM protocol.

26.4 Lotus Expeditor Client management

Device Manager supports management of Eclipse-based products, such as Lotus Expeditor Client. There is additional support for managing Eclipse features, Eclipse plug-ins, Eclipse preferences, and properties. The Device Manager agent in Lotus Expeditor Client is called the Enterprise Management Agent (EMA). The agent runs an OSGi R4 framework and its runtime accepts OSGi R3 and OSGi R4 bundles.

Device Manager is leveraging the BaseOMA DM stack and OSGi functions for the Enterprise Management Agent. In addition, the BaseOMA DM protocol is extended to support added function not defined by the BaseOMA DM standard, such as automatic feature and bundle prerequisite loading and tuning for the prerequisite resolution for OSGi bundles.

26.5 Architecture

Figure 26-1 on page 682 shows different components of the device management services in Lotus Expeditor. Each component is briefly explained.
26.5.1 Device Manager

The Device Manager is a set of servlets executing in the application server. Device Manager executes the management tasks or jobs on the connected devices.

Device Management Server includes a DM servlet that ensures that the device is enrolled with the server. For enrolled devices, the DM servlet checks if there are any pending jobs for the device and processes them after prioritizing the jobs. The DM servlet processes the jobs using the device plug-ins that interact with the device.

Device plug-ins are responsible for device identification, communication with the device, processing jobs on the device, and high-level management of tasks. A plug-in is provided for different classes of devices and is extensible to support new devices types. The Device Manager support for a particular device depends on its operating system and not on the make or model of the device. Device manager provides plug-ins to support the following device types:

- OMA DM 1.1.2 and 1.2, such as Nokia Series 80 (9300 and 9500) and Nokia E Series (E60, E61 and E70)
- OSGi devices with the Device Manager OSGi agent and its derivatives; including Windows 32 devices and Linux devices
- Platforms that run the managed client environment (Lightweight Infrastructure) for use with Tivoli® Provisioning Manager.

The job types implemented by the plug-in for Lotus Expeditor are:

- **Bundle control** - performs actions on OSGi bundles.
- **Command script** - runs a BaseOMA DM command script to perform operations on target devices or to return device information. Using the command script job, administrators can build command jobs that are more complex than a custom command job.
- **Custom command** - builds a free-form command. The job issues one or more specified commands to perform operations on target devices or returns device information.
- **Device configuration** - changes configuration parameters for devices.
- **Eclipse feature control** - performs actions on Eclipse features.
- **Eclipse feature distribution** - sends Eclipse features along with any required prerequisite Eclipse features to targeted devices.
- **Eclipse feature removal** - removes Eclipse features along with any prerequisite Eclipse features to targeted devices.
- **Eclipse preferences editing** - provides the ability to edit the Eclipse preferences under the tree at ./Configuration/Preferences/Eclipse.
- **Eclipse preferences retrieval** - retrieves Eclipse preferences under the tree at ./Configuration/Preferences/Eclipse.
- **Inventory collection** - scans devices for inventory data and stores that information in the database.
- **Native software distribution** - sends native software as an OSGi bundle along with any required prerequisite bundles to targeted devices and captures stdout, stderr, and return code messages.
- **Node discovery** - specifies a starting node in the BaseOMA DM tree and discovers all sub-nodes with a recursive search.
- **Properties editing** - provides the ability to edit properties under the tree at ./Configuration/Properties.
- **Properties retrieval** - retrieves properties under the tree at ./Configuration/Properties.
- **Run command** - provides the ability to run a command on the device.
- **Software distribution** - sends software as an OSGi bundle along with any required prerequisite bundles to targeted devices.
- *Software list update* - updates the client simulator tree with the list of available software that is registered in the Device Manager database. The request typically is initiated by the device, so the server retrieves the list and sends the list to the device.

- *Software removal* - removes software from targeted devices.

### 26.5.2 Enterprise Management Agent

The Enterprise Management Agent resides on the Lotus Expeditor Client. The Enterprise Management Agent is responsible for executing the commands sent by the Device Management server. The client device connects and interacts with the Device Management Server using the OMA DM protocol. However, the Enterprise Management Agent does not need to be connected to the server all the time. When the agent connects, the server identifies the client and executes the jobs pending for this device.

### 26.5.3 Device Manager database

The Device Manager database is a relational database that stores all the device management information. This database stores device-specific information, including the jobs for the devices. The Device Manager database is created with all the necessary tables, views, and queries.

### 26.5.4 Device Manager Console

The Device Manager console is a Java program that you can use to administer and monitor the Device Management Server. The console provides functionality to register and view devices, create jobs for viewing or modifying device or software properties, view the status of the jobs, and so on.

The Lotus Expeditor server installation and configuration program will install the console on the Lotus Expeditor server machine but the Device Manager console can also be downloaded from the Device Management Server and installed on any system. This allows administrators of the devices to control devices from various terminals and not necessarily on the server. The Device Manager console requires a DB2 database client for connecting to the Device Management database.

The first time you start the Device Manager console, it must be able to connect to the Device Management Server to log in and retrieve the properties file. The Device Manager console creates a backup of the properties file and subsequent startup will not need to connect to the Device Management Server for login and will connect directly to the Device Manager database for administration.
26.6 Lotus Expeditor Client sample scenario

This scenario shows how to distribute software to Lotus Expeditor Client using Device Manager provided by Lotus Expeditor Server. We show you how to use Device Manager to configure software distribution and software removal jobs for deploying Eclipse features.

Before we begin with the scenario Lotus Expeditor Server must be installed and configured, including Device Manager Server.

26.6.1 Enrolling the Lotus Expeditor Client in Device Manager

In order to use Device Manager to manage the Lotus Expeditor Client, you have to enroll the client with the Device Manager server. For example follow these steps:

1. Make sure your Device Manager server is started.
2. Start your Lotus Expeditor client and select File → Preferences.
3. Select Enterprise Management Agent and enter the details for your Device Manager account, as shown in Figure 26-2. You can click Test Connection to check for the server availability. Click OK to save the changes.

Note: The Device Manager console connects to the Lotus Expeditor Subscription Manager to authenticate the user.
4. Wait while Lotus Expeditor Client connects to the Device Manager server and checks for updates and applicable jobs. See Figure 26-3.
The first job ran by Device Manager in your Lotus Expeditor Client is the inventory collection job. Basically the Enterprise Management Agent will take an inventory of the device, for example OS, architecture, etc. This inventory is stored in the Device Manager database. You can see your enrolled device using the Device Manager console. For example follow these steps:

1. Open the Device Manager console. Log in as an administrator user (Figure 26-4).

2. Click Devices, select Return Anything and click OK. You will your Lotus Expeditor Client listed (Figure 26-5).
26.6.2 Software distribution for Lotus Expeditor Client

Distributing software to devices is the most common application of device management. Software for Lotus Expeditor Client is distributed as Eclipse features.

Distributing software requires you to:

1. Create and publish an update site with the feature to distribute.
2. Register the software with the Device Management Server.
3. Create a job for distributing it.

Creating and publish the update site

As you know, a feature is a way of grouping and describing different functionality that makes up an application. Grouping plug-ins into features allows the application to be installed and updated using an update site.

Device Manager supports the distribution of features, not entire update sites. This means that you have two options to package your composite application:

- You can package all the application plugins into one feature and then publish it in an update site. The drawback is that you cannot manage the application update at plug-in level, so you would need to redeploy the entire application to update a component.
- You can package each application plug-in, in one feature, and then declare your application feature as a composite feature, including by this means all
the other required features. In this scenario the application composite feature descriptor is shown in Figure 26-6 with all the other needed features included.

Figure 26-6   Application composite feature

Once you have the update site you have to publish in a HTTP Server, for example IBM HTTP Server. The update site should be available for HTTP access as shown in Figure 26-7.

Figure 26-7   Application update site

You can find the project interchange for the application delivered in this scenario in additional materials, see Appendix B, “Additional material” on page 721.
Registering the software with Device Management Server

An administrator places the update site on an IBM HTTP Server, then uses the Device Manager console to register the software with Device Manager so that it is available for distribution to Lotus Expeditor Clients with a software distribution job.

1. Open the Device Manager console. Log in as an administrator user.

2. In the left panel, see Figure 26-8, right-click Software and select New Software.

![Figure 26-8 Selecting the New Software option](image)

3. In the New Software properties window (Figure 26-9 on page 691), enter the following:
   - Software type: Since we are distributing software for Lotus Expeditor, select Eclipse feature.
   - URL: Fully qualified URL for the update site where the application composite feature is located. For his scenario the URL is http://xpd61.itso.ibm.com/com.ibm.itso.swtsample.application.updatesite/features/com.ibm.itso.swtsample.application.feature_1.0.0.jar

Note: For information about how to create an update site in Rational Application Developer and publish it using IBM HTTP Server, see Chapter 15, “Exporting components and publishing the update site” on page 375.
– Click **Fetch**. The feature name, version, and description are retrieved from the feature descriptor file as shown in Figure 26-9. If you want to, you can edit this field.

– Click **Next**.

![New Software Properties: Software](image)

Figure 26-9   New software properties

4. Select all the device class and job type combinations you require (Figure 26-10 on page 692). For this scenario, select, at least, **Win32 - Eclipse Feature Software Distribution**. Click **OK**.
5. The software will be registered (Figure 26-11). To check it, click the **Software** tab and select **Return Anything**.
Creating a software distribution job
With the Device Manager console, you can create jobs for specific devices, user groups, device classes, enrolled devices, and so on.

We will create a software distribution job for all the Lotus Expeditor Clients to install a specific Eclipse feature.

To create a software distribution job, perform the following tasks:
1. In the Device Manager console, right-click Jobs, and select Submit Job.
2. Select the following options in the Target Devices window (Figure 26-12):
   - Device class: Since this is a software distribution for Lotus Expeditor Client running on Windows, select Win32.
   - Query: For this example, we will distribute the software to All Win32 devices.
   - Owner group: Select an option if you want.
   - Select whether or not the job is for enrolling or already enrolled users. In this example, it is for Both enrolling and currently enrolled.
   - Click Next.

Figure 26-12   Target devices
3. In the Submit Job: Attributes pane, Figure 26-13, select **Software Distribution** as the job type. Enter adequate dates for activation and expiration fields, priority, and a description for the package. Click **Next**.

4. In the Submit Job: Job Parameters pane, Figure 26-14 on page 695, Click **Add Group** and select the software to be distributed (the Eclipse feature registered in “Registering the software with Device Management Server” on page 690). Click **Next**.
5. Review the job summary and click **Ok**. See Figure 26-15.

6. You will be shown a job submission status (Figure 26-16 on page 696). Note that there are already one eligible target device. This is the Lotus Expeditor Client you enrolled in “Enrolling the Lotus Expeditor Client in Device Manager” on page 685. Click **Close**.
7. The job has been created successfully. To see the job, click Jobs (Figure 26-17), select Return Anything, and click OK.

**Figure 26-17  Job registered**

**Checking job progress on the server**

To check the progress of a job on the server, perform the following steps:

1. To verify the status of your job, click Jobs. Click Target Device Class. Select Return anything with a device class that matches. Select Win32, and click OK.

2. You should be able see the job in your list. Right-click the job select View Job Progress and click OK. This will show the current status of the job as shown in Figure 26-18.
checking job progress on the client

Checking job progress on the client involves verifying if the intended job has been executed, in this case if the application has been installed. Figure 26-19 shows the application installed by Device Manager.

26.6.3 Software removal for Lotus Expeditor Client

With the software removal job, you can remove software that was distributed as Eclipse features. This job does not update the device software inventory in the
Device Manager server. To update the Device Manager server, you have to run an inventory collection job for this device.

Here, we create a software removal job for all Lotus Expeditor Clients running on Windows that remove the recently installed application.

To create a software removal job, perform the following steps:

1. In the Device Manager console, right-click Jobs and select Submit Job.
2. Select the following options in the Target Devices window:
   - Device class: Win32.
   - Query: All Win32 Devices
   - Owner group: Select an option if you want.
   - Select whether or not the job is for enrolling or already enrolled users. This example is for currently enrolled devices.
   - Click Next.
3. Select Eclipse Feature Software Removal as the job type. Enter adequate dates for activation and expiration fields, priority, and a description. Click Next. See Figure 26-20 on page 699.
4. Click **Feature Control Step** and click **Add Group**. Select **Uninstall** as Action. Then, select the feature to be removed (for this example, the feature distributed in “Creating a software distribution job” on page 693). Click **Next**.
5. Review the job summary and click **OK**.

6. You will be shown a job submission status. Click **Close**.

7. The job has been created successfully. To see the job, click the **Jobs** tab, select **Return Anything** and click **OK**. See Figure 26-22 on page 701.
26.6.4 Software update for Eclipse features

What do you do when a new feature becomes available or a bug is fixed for a particular application that has been distributed as an Eclipse feature? Updating a feature or a plug-in basically involves creating a new feature with the updated files and registering this feature with the Device Manager. A Software Distribution job can then be submitted for the device where you want to install the updates. On a Lotus Expeditor client, the files for a feature or plug-in are always stored in a subdirectory whose name includes a version identifier. Different versions of a feature or plug-in are given unique identifiers to ensure that multiple versions of a feature or plug-in can coexist on a disk. This means that installing or updating features always requires adding files, but never requires deleting or overwriting existing files.

26.6.5 Client-side troubleshooting

Robust logging mechanisms are provided with Lotus Expeditor to help you debug any problems that you may encounter. See Section 26.7, “Troubleshooting with Device Manager” on page 702.
26.7 Troubleshooting with Device Manager

There could be situations where you might need to view the Device Management Server traces to check the status of the server and the jobs submitted. By default, the tracing is disabled. The trace log is primarily a development tool and should be enabled when troubleshooting.

Enabling the trace log can affect your system performance. Extended use of trace logging can also consume large amounts of disk space. To manually enable tracing, do the following steps:

1. Navigate to the directory
   DMS_HOME\config\dmserver.war\WEB-INF\classes. Locate the
   traceConfig.properties. To start the trace facility for one or more of the Device Management components, you will need to edit the traceConfig.properties.
   Open this file in a text editor.
   
   a. You can select the components that you want to trace by setting the corresponding parameter to true. For example, if there is a problem with plug-ins, you could set the traceEnabled.plugins to true. See Example 26-1.
   
   b. You can adjust the amount of trace data by changing the value for the TraceLevel parameter. The default is 1 (least amount of data).
   
   c. You can specify the file size and number of files to store trace data. The default is three files of 512 KB each.

Example 26-1 Flags for different components of Device Manager

```
#trace enabled for components
cOMPONENT.conSOle=true
cOMPONENT.dmserver=true
cOMPONENT.event=true
cOMPONENT.notification=true
cOMPONENT.plugins=true
cOMPONENT.resultscollector=true
cOMPONENT.twgapi=true
cOMPONENT.database=true
cOMPONENT.enrollserver=true
cOMPONENT.datconverter=true
cOMPONENT.mcollect=true
cOMPONENT.notificationhandler=true
cOMPONENT.api=true
cOMPONENT.userservices=true
cOMPONENT.federator=true
```

2. Save your changes.
3. You can start the trace facility with the new setting by doing one of the following actions:
   a. Restart DMS_AppServer in WebSphere Application Server.
   b. In a browser, type the following Web address:
      http://DMS_host_name/dmserver/TraceServlet?trace=set. If it was successful, you get a “SUCCESS!” message in the browser.
   c. Run the following command:
      
      DMS_HOME/bin/server.sh -app dmserver -trace set

4. Perform the device management tasks. Review the trace information in the TraceDMSn.log file.
Developing Notes applications

This appendix describes how to build a Notes application that was used in the case study scenario. Here is the description for building:

- Notes discussion forum
  Used as a sample of existing Notes applications that can be enhanced and turned into a NSF component for composite applications.

You can either use this description to build the applications or download the provided code.
Notes discussion forum

In the use case scenario we assume an existing Notes Discussion Forum that we extend to become a Notes Component that can be used in a composite application. The options you have are either to use the existing NSF component provided as download or create it from scratch.

This section provides step-by-step guidance to create this Notes database by using the default discussion template and outlining the changes we have made to add details about Customer ID and Reservation ID.

Software required

To create a new discussion database, you have to install Lotus Notes 8. It provides the required functionality to create a new Notes application.

For making the modifications, you have to make sure that you have Lotus Domino Designer 8 installed on your system.

Creating a new discussion database

The first thing you want to do is to create a new discussion database. For this purpose there is a template in Lotus Notes available already. Use this to create a new discussion database call Discussion Forum.NSF.

1. If you do not have Lotus Notes 8 running, open it now.
2. By default you should see two tabs in the workbench. One of them should be Home which you need to select.
3. Select File → Application → New to open up the New Application dialog.
4. Keep the Server selection to Local and define the Title for your application as “Discussion Forum”.
5. The File Name section should automatically create the file name using the information you just entered in Title.
6. Next you should click the Encryption button that opens up the encryption settings dialog. Select Do not locally encrypt this database as shown in Figure A-1 on page 707. Then click OK to close the dialog again.
7. Back in the previous dialog you now have to specify the template that you want to use for the new application. By default the Server entry should show Local if not use the drop-down list to go back to Local.

8. Below you see a list of different templates. Select the Discussion - Notes & Web (7) template.

9. At the bottom of the dialog deselect the check box that says Inherit future design changes. By doing this your new discussion forum will not inherit changes from this template. If you do not deselect, the changes you make to this database might be overwritten.

Make sure that your New Application dialog looks like the one shown in Figure A-2 on page 708.
Figure A-2 Create a new discussion forum

10. Close the dialog by clicking OK and the new discussion forum is created and opened in Lotus Notes.

You have created a new discussion forum which is ready to go. You can add entries and reply to them. There are different views such as By Author or By Category. Since you just created the database there are no entries and it looks like Figure A-3 on page 709.
Adding additional features to the discussion database

The second step is to enhance the default structure of the discussion database to allow for car rental specific items. In this scenario the discussion database is used for discussing matters regarding certain customers and their reservations. Based on that we want to have the ability add entries which have a customer ID as well as a reservation ID and be able to sort them accordingly using these IDs.

Extending the Main Topic and Response Form

1. In order to make any design changes in the discussion database you need to launch Lotus Domino Designer 8. This gives you access to the database design elements.

2. First you need to open the Discussion Forum you previously created. Click File → Application → Open and the Open Application dialog appears. As shown in Figure A-4 on page 710, select Discussion Forum.NSF and click Open. This opens up the discussion database in the design mode.
What you want, to have in the discussion forum, is to be able to add new discussion threads. While doing that you want to provide additional information which is currently not covered by the default template of the discussion database. The additional information is the customer's ID as well as the reservation ID. Those two new fields you add to the Main Topic as well as to the Response view.

3. In Lotus Domino Designer you should see a pane on the left. Select **Forms** and **Main Topic**. As a result the right pane should change and show the details for the Main Topic. Scroll down until your screen looks like Figure A-5 on page 711.
Figure A-5  Domino Designer showing Main Topic Form

This the default Main Topic Form as defined in the discussion database template. Now you want to add the fields for customer and reservation ID.

4. Click into the table cell in the upper half of the right pane that says Attach a file: in order to set the focus.

5. Add two rows to that table by right-clicking and selecting Insert → Row. Repeat this once more.
Next you will copy the categories field from the cell above the rows you just created and rename them.

6. Click into the NewCats (not the Categories) field above the two rows you just created and right-click. In the menu, select **Copy**.

7. In the first and second row select the second cell from left one after another and paste the field by right-clicking and selecting **Paste**.

8. Select the first field you just pasted into a cell and right-click. Select **Field Properties** and the properties view appears.

9. Change the Name to **CustomerID** and deselect **Allow multiple values** as shown in Figure A-6. Close this dialog and do the same for the second field. Here you enter **ReservationID**.

![Field Properties](image)

**Figure A-6**  **CustomerID** and **ReservationID** fields

10. By default for whatever reason the settings for these fields is that they will not show in Notes R4.6 or later. In order to make them visible, select the hide tab and deselect **Hide paragraph from Notes R4.6 or later** for both customerID and reservationID as shown in Figure A-7.
11. In the first column, next to the fields type in "Customer ID:" and in the next row "Reservation ID:". Your form should now look like the one shown in Figure A-8.

12. In order to make this text visible repeat Step 10 once you have typed the text.

Most likely you do not just want to add a main topic to the discussion base but also a response to an entry. That means we want to update the response form.
with these two fields as well. Since this is a Subform you have to change this in the shared code section.

13. In the left pane scroll down and select **Shared Code → Subforms → RespBanner**. This brings up the Response to Main Document up in the right pane.

14. Repeat Steps 4 - 12 for this form in the same manner until your Response Form looks like the one shown in Figure A-9.

Figure A-9  Update Response Subform

The CustomerID and ReservationID fields in the Response Subform will be updated with the details you entered in the Main View when creating a discussion entry. In order to access to this information you need to make some additional changes.

15. Select the **CustomerID** field. At the bottom you should see two panes. In the right pane type **CustomerID**.

16. Repeat the same for the **ReservationID** field and enter **ReservationID**. It should look like Figure A-10 on page 715.
Figure A-10  Value for the ReservationID field

17. Do not forget to save your work from time to time pressing Ctrl → s on your keyboard.

Adding By Customer ID and By Reservation ID Views
To sort the discussion forum by customerID or reservationID you need to add new views for each selection. You also need to define the way these views should sort the content.

1. On the left pane in Lotus Domino Designer click Views. The right pane will be updated to show all currently available views.

2. At the top of the right pane click New View which opens the Create View dialog.

3. As the new View name you need to enter “By Customer ID”. Leave all other parameters as shown in Figure A-11 on page 716 and click Save and Customize which creates the new view.
4. Once created, a new information box appears. The Name should show “By Customer ID”, and you need to enter the same into the Alias field underneath.

5. Do not close the information box. By default the first column should be select and you can switch to the column properties using the drop-down list at the top of the information box.

6. The default Title for the first column is Date. Remove the text from the Title field.

7. Change the Width to 2 and deselect Resizable. This is required because it is a categorized column and will stick past the other columns. When it has something in it, it takes up the whole row. When it does not, it is empty.

8. At the bottom select Show twistie when row is expandable.

9. In the information box select the second tab which is responsible for sorting and adjust them to match what is shown in Figure A-12 on page 717.
10. Now you also need to assign a name for the column. For this click the **Advanced** tab (last one on the right) and enter **CustomerID** as the **Name**.

11. In the same tab at the bottom, deselect **Show values in this column as links** as shown in Figure A-13 on page 718.

---

**Figure A-12**  Settings for first column in Customer ID view
12. Now you need to add a view for sorting by Reservation IDs. For that go through the same Steps 1 - 5 as for the sorting by Customer ID view. Once completed continue with Step 13.

13. You should be in the info page and the first tab selected. Make sure that from the drop-down list at the top you select **Column**.

14. Remove the default text shown in the **Title** field, deselect **Resizable** and select **Show twistie when row is expandable**.

15. Go to the second tab which defines the sorting criteria. Make sure to have the same settings as in Figure A-14 on page 719.
16. Select the last tab which contains the advanced settings. Define the Name as ReservationID. Using this name the view can be accessed programmatically. You will need that later on.

17. Deselect Show values in this column as links.

18. Close the information box and save your new views.

This was the last step in creating the discussion forum for the car rental use case scenario. You should be able to add new discussion threads in which you can define a customer and reservation ID. You have created two additional views, which sort by customer or reservation ID.
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 to:

ftp://www.redbooks.ibm.com/redbooks/SG247367

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 Redbooks form number, SG247367.

Using the Web material

The additional Web material that accompanies this book includes the following files:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sg247367.zip</td>
<td>Zipped Code Samples</td>
</tr>
</tbody>
</table>
System requirements for downloading the Web material

The following system configuration is recommended:

- **Hard disk space:** 100 MB minimum
- **Operating System:** Windows
- **Processor:** 2GH or higher
- **Memory:** 2GB or higher

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 IBM Redbooks” on page 724. Note that some of the documents referenced here may be available in softcopy only.

- **IBM WebSphere Everyplace Deployment V6 Handbook for Developers and Administrators Volume II: Smart Client Application Development**, SG24-7183

Other publications

These publications are also relevant as further information sources:


Online resources

These Web sites are also relevant as further information sources:

- Lotus Expeditor InfoCenter
- IBM Education Assistant for Expeditor
Lotus Expeditor and SOA

Developerworks - Lotus Expeditor Forums

Composite Applications blog

Lotus Expeditor support site
http://www.ibm.com/software/lotus/expeditor/support/

IBM WebSphere Integration Reference Architecture
http://www.watchit.com/websphere

Business Integration

IBM WebSphere Software
http://www.ibm.com/software/websphere

SOA
http://www.ibm.com/developerworks/webservices/newto/

Model Driven Architecture

Business Performance Management
http://www.ibm.com/software/info/topic/perform/

Eclipse
http://www.eclipse.org

Lotus Expeditor Car Rental Customer Service demonstration

How to get IBM Redbooks

You can search for, view, or download Redbooks, Redpapers, Hints and Tips, draft publications and Additional materials, as well as order hardcopy Redbooks or CD-ROMs, 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
Abstract Window Toolkit (AWT) 43, 71, 283, 550
action handler 47, 269, 302
action name 90, 123, 160, 555, 632
action parameter 121, 573
action value 118, 581
ActionHandler 277, 303
ActionRequest request 94, 96, 128, 161, 491
ActionResponse response 94, 128, 161, 491
ActiveX control 323
application 73, 171, 305, 360
application programming interface (API) 40, 70, 199
application role 399, 441, 470, 526
available portlets 77, 654
AWT component 283, 288
    Property Broker 286
    queues events 288
AWT Frame 286, 298
AWTMessage component 294
    source code 294
    WSDL file 301

B
BaseOMA DM
    command script 683
    protocol 681
    standard 681
tree 683
basic portlet 102, 112, 119, 157, 490
binding name 89, 123, 185, 273, 292
booking component 86, 478
boolean rc 228

C
CASAMPLE folder 394
    com.ibm.itso.casample.updateSite directory 396
    exported artifacts 395
client platform 39–40, 70, 502, 549
code snippet 201
com.ibm.itso.casample.sendMessage.feature 412,
com.ibm.itso.compapp.carrental.userreservations.feature 512
component communication 49, 82, 109, 115, 251, 306
component palette 599, 619
    new components 653
composable component 548, 615, 674
    ActiveX Controls 323
    approach 610
    AWT/Swing components 283
    AWTMessage component 297
bundle 171
component 261, 324, 363, 377, 547, 608, 610
Components 548
components tie 34
composer 44
collection 44
definition 541
design 351
desire 41
development 81, 538
different types 674
Discussion Forum 560
Eclipse view 43
editor 112, 350, 548, 597–598, 605, 618, 674
environment 645
feature 38, 549
First page 460
important characteristics 86
instance 400, 404, 443
instantiation 145
instantiation process 144
Java Runtime Environment 286
launch option 262
layout 44, 153
main idea 116
new design elements 610
NSF component 705

© Copyright IBM Corp. 2007. All rights reserved.
<table>
<thead>
<tr>
<th>Index</th>
<th>729</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Definition</strong></td>
<td>73</td>
</tr>
<tr>
<td>Extensions tab</td>
<td>101, 172, 260, 511</td>
</tr>
<tr>
<td><strong>F</strong></td>
<td></td>
</tr>
<tr>
<td>Feature</td>
<td>14, 38, 52, 70, 86, 264, 290, 350, 375, 411, 466, 478, 507, 614, 681</td>
</tr>
<tr>
<td>Feature project</td>
<td>510–511</td>
</tr>
<tr>
<td>Feature.xml file</td>
<td>512</td>
</tr>
<tr>
<td>Dependencies tab</td>
<td>512</td>
</tr>
<tr>
<td>File extension</td>
<td>162</td>
</tr>
<tr>
<td>First page</td>
<td>234, 458</td>
</tr>
<tr>
<td>Layout mode</td>
<td>463</td>
</tr>
<tr>
<td>FormAttachment</td>
<td>215</td>
</tr>
<tr>
<td>FormData</td>
<td>215</td>
</tr>
<tr>
<td>Frameset</td>
<td>570, 614</td>
</tr>
<tr>
<td>Function addView</td>
<td>368</td>
</tr>
<tr>
<td><strong>H</strong></td>
<td></td>
</tr>
<tr>
<td>HTTP Server</td>
<td>349, 376, 516, 608, 618, 676, 689</td>
</tr>
<tr>
<td>Directory</td>
<td>516</td>
</tr>
<tr>
<td>Document root directory</td>
<td>396</td>
</tr>
<tr>
<td>Root document folder</td>
<td>396</td>
</tr>
<tr>
<td>Update site</td>
<td>615</td>
</tr>
<tr>
<td>Import org.eclipse.swt.SWT</td>
<td>228, 256, 328</td>
</tr>
<tr>
<td>Information box</td>
<td>716</td>
</tr>
<tr>
<td>Input message</td>
<td>89, 124, 186–187, 273, 293</td>
</tr>
<tr>
<td>Input name</td>
<td>95, 128, 162</td>
</tr>
<tr>
<td>Input property</td>
<td>45–46, 93, 118, 155, 194, 269, 272, 553</td>
</tr>
<tr>
<td>Int i</td>
<td>141, 206</td>
</tr>
<tr>
<td>Integrated Development Environment (IDE)</td>
<td>82</td>
</tr>
<tr>
<td>Inter-component communication</td>
<td>49, 82, 551, 562, 609, 618</td>
</tr>
<tr>
<td>Additional topics</td>
<td>49</td>
</tr>
<tr>
<td>Inter-portlet communication</td>
<td>47, 100, 102, 192</td>
</tr>
<tr>
<td>IPPageLayout.RIGHT</td>
<td>177, 306, 369</td>
</tr>
<tr>
<td>IPerspectiveFactory</td>
<td>368</td>
</tr>
<tr>
<td>ITSO Calcium</td>
<td>460</td>
</tr>
<tr>
<td>ITSO Car Rental</td>
<td></td>
</tr>
<tr>
<td>Application</td>
<td>334</td>
</tr>
<tr>
<td><strong>Database</strong></td>
<td>628</td>
</tr>
<tr>
<td>Database feature</td>
<td>646</td>
</tr>
<tr>
<td>Project interchange file</td>
<td>336</td>
</tr>
<tr>
<td>Reservation</td>
<td>534</td>
</tr>
<tr>
<td>Reservations application</td>
<td>533</td>
</tr>
<tr>
<td>Reservations Template</td>
<td>526, 529</td>
</tr>
<tr>
<td>Sample application</td>
<td>86, 116, 477</td>
</tr>
<tr>
<td>Sample composite application</td>
<td>116</td>
</tr>
<tr>
<td>Scenario</td>
<td>334</td>
</tr>
<tr>
<td>ITSO Car Rental Reservations Template</td>
<td>517</td>
</tr>
<tr>
<td>ITSO Composite Application</td>
<td></td>
</tr>
<tr>
<td>CrossWire Sample</td>
<td>458</td>
</tr>
<tr>
<td>Sample</td>
<td>410</td>
</tr>
<tr>
<td>Sample template</td>
<td>444</td>
</tr>
<tr>
<td>ITSO Composite Application Sample</td>
<td>396, 405, 407</td>
</tr>
<tr>
<td>Template</td>
<td>426, 444, 458</td>
</tr>
<tr>
<td>ITSO Composite Application Sample template</td>
<td>404</td>
</tr>
<tr>
<td>ITSO user</td>
<td>430, 453, 472</td>
</tr>
<tr>
<td>Reservation</td>
<td>517</td>
</tr>
<tr>
<td>Reservations page</td>
<td>518</td>
</tr>
<tr>
<td><strong>J</strong></td>
<td></td>
</tr>
<tr>
<td>J2SE 5.0</td>
<td></td>
</tr>
<tr>
<td>Win32 x86</td>
<td>291</td>
</tr>
<tr>
<td>Win32 x86 runtime</td>
<td>309</td>
</tr>
<tr>
<td>Java Native Interface (JNI)</td>
<td>199</td>
</tr>
<tr>
<td>Java Runtime Environment (JRE)</td>
<td>284</td>
</tr>
<tr>
<td>java.io.IOException</td>
<td>94, 128, 161</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>314</td>
</tr>
<tr>
<td>javax.portlet.PortalSessionUtil</td>
<td>314, 361</td>
</tr>
<tr>
<td>JSP</td>
<td>74, 93, 127, 155, 491</td>
</tr>
<tr>
<td>JSP file</td>
<td>160</td>
</tr>
<tr>
<td>File extension</td>
<td>162</td>
</tr>
<tr>
<td>JSP page</td>
<td>95, 491</td>
</tr>
<tr>
<td><strong>K</strong></td>
<td></td>
</tr>
<tr>
<td>KeyWordType ty</td>
<td>238</td>
</tr>
<tr>
<td><strong>L</strong></td>
<td></td>
</tr>
<tr>
<td>LabDescription.setLayoutData</td>
<td>231</td>
</tr>
<tr>
<td>Launch menu</td>
<td>73, 400, 443, 458</td>
</tr>
<tr>
<td>Select Administration</td>
<td>400</td>
</tr>
<tr>
<td>Select Templates</td>
<td>458</td>
</tr>
<tr>
<td>Lotus Expeditor</td>
<td>31, 69, 86, 153, 197, 251, 283,</td>
</tr>
</tbody>
</table>
Building Composite Applications

AWT components 285
base installation 352
component communication 71
composable components 43
composite application 43, 324, 508
existing AWT application 286
initial load 112
intercommunicate two components 48
Lotus Notes 8.0 version 6.1.1 549
main requirement 154
Network Client Installer 645
powerful and easy-to-use Smart Client applications 222
property broker 88
Lotus Expeditor client 38, 40, 70, 198, 263, 287,
311, 349, 351, 413, 441, 473, 502, 638, 679, 681
composite application 474
Device Manager agent 681
network connection 431
new composite application 441
platform 42, 49, 431, 504
side 43
software distribution 688
software distribution job 693
software removal 697
software removal job 698
start 345
Lotus Notes 79, 544, 547, 592, 609
composable components 550
Lotus Notes 8 538, 547, 597–598, 607, 673, 706
composite applications 548
Home Portal Account 598
major new enhancement 538
new application 609
new architecture 547
premium client 549
standard version 641
update site 627
WebSpere Portal 639

M
mainView.updateMessage 272
mainView.updateMessage 303
Managed composite application
various pieces 349
MANIFEST.MF file 171–172, 255, 286, 340, 485
matching rule 360, 412, 466, 521
menu bar 156, 253, 289, 331, 377, 512
Run command 309
Run option 331
message name 89, 123, 185, 273, 292
messageLabel.setText 257, 289
Model View Control (MVC) 103, 488
business logic 103
ModifyEvent text (MODIFYTEXT) 229
multi-line text 202
multipage application 471

N
Name field
type 583
type DiscussionForumListClient 587
name space 267, 563
Network Client
Install 399
Installation 40, 349
Installer 42, 272, 376, 431, 443, 501, 638
Installer component 645
non-connected environment 35, 41
composite application 35
Notes action 541, 550, 560
same names 581
Notes application 538, 548, 600, 606, 674, 705
Composite Application XML 607
Design elements 566
local copy 646
other elements 550
sufficient access rights 538
Notes URL 620
NSF 547, 597, 606–607, 610, 674
NSF component 539–540, 547–548, 600, 610,
674, 705
Additional information 645
design 561
inter-component communiction 551
local Portles 549
Notes actions 581
support 639
view 548, 675
visual representation 548
NSF file 606–607
new design element 607
NSF view 614
component 612
component change 613, 676

O
Object Linking and Embedding (OLE) 323
Object Technology International (OTI) 198
OLE Document 323
OLE Object 324
OleClientSite 324
OleControlSite 324
OMA DM
  1.2 implementation 681
device agent 681
  Management Server 681
overview 681
protocol 681
Open Mobile Alliance (OMA) 680
Open Service Gateway initiative (OSGI) 70, 352, 680
Open Services Gateway initiative (OSGI) 311, 334
open source
  database 629
  widget toolkit 198
operating system
  color palette 226
  resource 220
operation name 89, 123, 185, 273, 292
OSGi bundle 70, 286, 334, 681
  prerequisite resolution 681
OSGi Event Admin
  API 334
  mechanism 333
OSGi framework 73, 154, 334
  Eclipse 3.2.1 implementation 334
output message 89, 123, 184–185, 274, 292
output property 45–46, 93, 117, 155, 265, 272, 540, 559, 563
  WSDL file 274
Owner ID 110, 314

P
Page property 406, 459, 517, 635
pageLayout.addS tandaloneView 177, 262, 305, 330
part name 89, 123, 185, 273, 292
Personal Information Management (PIM) 554
plugin.xml 73, 110, 174, 260, 511
plugin.xml file 73, 93, 100, 155, 174, 260, 275, 286, 302, 331, 360, 488, 512
created wire 192
extension point 260
property broker wire extension point
com.ibm.rcp.propertybroker.PropertyBroker-Wire 100
wire definitions 195
port type
  CustomerIdentification_Service 123
  CustomerReservations_Service 124
  name 90
  name CustomerIdentification_Service 123
  name CustomerReservations_Service 124
  name ReceiveMessageService 274
  name ReservationDetails_Service 125
  name Wire_Service 184
  receiveAuth 90
  ReceiveMessageService 274
  ReservationDetails_Service 126
  Wire_Service 184
port type name
  Wire_Service 186
portal server 31, 36, 70, 133, 371–372, 501, 598, 606, 674
  assembled composite application XML file 513
  domain name 78
  new template log 145
  new user 529
  new users 528
Portal tool 118, 175, 272, 274, 376, 399–400, 473, 508, 597, 606, 651
Portal-managed environment 40, 79, 100, 503
  portal server 503
  action actionNameParameter 123
  action name 89, 185, 273, 292
  component specific parameters 354
  param boundTo 123
  param name 89, 185, 273, 292
  portlet shell 119
  preferences 97
  prior step 120
portlet application 69, 116, 132, 154, 515
portlet component 91, 115, 153, 253, 291, 359, 370, 387, 400, 443
  Configuration options 359
  portlet communication 91

Index 731
portlet communications 154
sample composite application 92, 117
significant advantage 116
SWT component intercommunicates 387
WDSL file 95
portlet name 73, 102, 119–120, 192, 370–371
portlet project 112, 119, 156, 387
Portlet Wiring Tool 49
portType name 89, 123, 185, 273, 292
processAction 47, 93, 117, 155, 490
Project Explorer
  pane 120
  view 97, 183
property broker 41, 86, 117, 154, 182, 252, 286, 301, 311, 438, 491, 551, 569
  action handler 48
  action requests 142
  customer ID 117
  entry point 91
  existing discussion forum functionality 569
  existing discussion forum 571
  input property 560
  plugin extension 190
  propertyChanged event 47
  RCP component 48
  received action 95
  wire definitions 100
Property ClassName 314
property name 111, 121, 267, 553
  constant definitions 267
Property NS 314
PropertyBroker 116
PropertyBrokerWire 100, 190
provisioning URL
  plug-in components 466
proxy portlet 355, 361, 387
public void 160, 209, 262, 288, 329, 369
  actionPerformed 296
  createPartControl 259, 298, 328
  doView 95, 129
  handleEvent 343
  mouseUp 109
  processAction 94, 128
  run 271, 288, 341
  runWithEvent 104, 271, 303
  sendMessage 300
  setFocus 260, 329
  start 339
  updateMessage 270, 304
widgetDefaultSelected 257
widgetSelected 257
R
Rational Application Developer
  update site 690
  WAR file 132
Rational Application Developer (RAD) 90, 118, 153, 251, 312, 335, 375–376, 502, 690
RCP component 47–48, 103, 105, 197, 251, 358, 362, 399, 422
  action handler 269
  composite application 437
  property broker 264
RCP view 172, 253, 298, 333, 359
RCPUsers role 436, 450, 471
rd 95, 129, 161, 490
rd.include 95, 129, 161, 490
ReceiveMessage portlet 163, 253, 292, 299, 393, 401
  portlet.xml file 188
  WSDL file 292
Remote portlet
  Sample 444
  Viewer 443
RenderRequest request 95, 129, 160
RenderResponse response 95, 129, 160, 490
request.getParameter 94, 128, 161, 491
request.getResponseContentType 95, 129
request.setAttribute 94, 128, 161, 490
required data item
  further processing 108
Reservation ID 117, 558, 565, 613, 676, 706
  same functionality 591
  Select Document 633
  SelectDocumentByReservationID 568
Rich Client 39, 355, 359, 400, 465, 504, 544, 600, 638–639, 673
  composite application 544
  SWT view 418, 525
  update site 600
  WSRP instance 506
Rich Client WSRP Viewer
  enablement portlet 506
  menu 448
right pane 574, 647, 710
  copy 575
  second half 579
Building Composite Applications
Building Composite Applications

Learn about composite applications and component intercommunication
This IBM Redbooks publication covers composite applications for desktop solutions that require multiple types of applications to run in a cooperative environment by providing intercommunication between components. Service-oriented architecture (SOA) composite applications deliver high levels of business services and this book covers the architecture, available tools, component considerations, as well as assembling, deploying and wiring components in WebSphere Portal and Rich Client Platforms, such as Lotus Expeditor and Lotus Notes 8.

Develop components using Rational Application Developer
Lotus Expeditor is a client platform for end-to-end smart client application solutions. Expeditor provides services to install and manage these applications and allows users to easily launch and switch among these applications. Expeditor leverages Eclipse technology to provide a client solution that runs on multiple operating systems. The benefits of composite applications include reducing the total cost of ownership for client solutions through reusability of existing components and the sharing of these components across applications, migrating existing applications to run on the client platform, controlling access to applications based on roles assigned to users and improving the user experience by allowing composite application components to easily communicate between them.

Assemble and deploy managed applications
This publication is targeted at architects, developers and administrators who need to understand the implemented architecture to develop, assemble and deploy composite applications in Lotus Expeditor V6.1.

For more information:
ibm.com/redbooks