Building Composite Applications in Lotus Expeditor V6.1

Assemble managed client applications in an SOA environment

Understand how RCP and Portlet components communicate

Learn how to deploy composite applications

Juan R. Rodriguez
Alex Barbosa Coqueiro
Rafael Sanchez
Robert Schneider

ibm.com/redbooks
International Technical Support Organization

Building Composite Applications in Lotus Expeditor V6.1

February 2007
First Edition (February 2007)

This edition applies to Version 6, Release 1 of Lotus Expeditor V6.1.

Note: This book is based on a pre-GA version of a product and may not apply when the product becomes generally available. We recommend that you consult the product documentation or follow-on versions of this redbook for more current information.
Contents

Notices ................................................................. vii
Trademarks ............................................................ viii

Preface ................................................................. ix
The team that wrote this IBM Redpaper. ........................................... ix
Become a published author ........................................................ xi
Comments welcome ........................................................ xi

Chapter 1. Composite applications ........................................... 1
  1.1 Overview ......................................................... 2
  1.2 Server based and managed composite applications .......... 2
  1.3 Features of a composite application ........................... 4
  1.4 Integration with WebSphere Portal .......................... 5
     1.4.1 Overview .................................................. 6
     1.4.2 Client Composite Application Infrastructure ........ 7
     1.4.3 A portal administered client .......................... 9
  1.5 Composite applications in Lotus Expeditor .................. 10
     1.5.1 Composable components in Lotus Expeditor ........ 10
     1.5.2 Component layout procedures ........................ 11
     1.5.3 Intercommunication between components ............. 12

Chapter 2. Components in composite applications ................. 17
  2.1 Overview ......................................................... 18
  2.2 Local Eclipse components ...................................... 19
  2.3 Client service portlets project .............................. 20
     2.3.1 Lotus Expeditor portlet support ........................ 20
  2.4 Client service Web project .................................... 22
  2.5 Remote portlets ................................................ 24
     2.5.1 WSRP services .......................................... 25
     2.5.2 Lotus Expeditor WSRP support ........................ 27
  2.6 Future potential components .................................. 27

Chapter 3. Managed applications ....................................... 33
  3.1 Overview ......................................................... 34
  3.2 Prerequisites for declaring composite applications ....... 35
  3.3 Access control and roles ...................................... 36
  3.4 Provisioning ..................................................... 38
  3.5 Definition and placement ...................................... 39
     3.5.1 Portlet instances versus portlet clones ............. 40
3.6 Providing components in a portal sense .......................... 41
  3.6.1 Placing Eclipse based components ................. 41
  3.6.2 Placing portlets as components .................. 46
  3.6.3 Placing a WSRP portlet viewer ..................... 47
  3.6.4 Placing a Web application as a component ....... 48
3.7 Specifying custom settings ................................. 49
3.8 Templates ................................................. 51
  3.8.1 Creating a template ............................... 52
3.9 Laying out applications programmatically ............... 53
  3.9.1 Implementing a perspective ....................... 54
  3.9.2 A sample remote portlet component ............. 57
  3.9.3 Declaring the perspective ......................... 59
  3.9.4 Declaring the application ......................... 59

Chapter 4. Component communication .......................... 61
  4.1 Overview ............................................. 62
  4.2 Declaring data items and actions ...................... 63
  4.3 Preparing components for inter-communication ....... 67
    4.3.1 Portlet components ............................. 67
    4.3.2 Rich Client Platform components .............. 80
  4.4 Declaring wires ...................................... 86
    4.4.1 Wires as part of your plugin.xml .............. 87
    4.4.2 Wires declared through WebSphere Portal ..... 89

Chapter 5. The ITSO Car Rental sample composite application .......................... 91
  5.1 Architectural overview ................................ 92
  5.2 The database component ................................ 94
    5.2.1 Customer model ................................ 95
    5.2.2 Car model ....................................... 96
    5.2.3 Contract model .................................. 97
    5.2.4 Database helper functions ....................... 98
  5.3 The booking component ................................ 99
  5.4 The car selection component .......................... 102
  5.5 The credit card component ............................ 108
  5.6 The customer contract list component ............... 108
  5.7 The weather component ................................ 111
    5.7.1 Tasks for producer portals ...................... 112

Appendix A. Additional material .............................. 115
  Locating the Web material ................................ 115
  Using the Web material .................................. 115
    System requirements for downloading the Web material 116
    How to use the Web material ........................... 116
**Related publications** .............................................. 117
IBM Redbooks .................................................................. 117
Other publications .......................................................... 117
Online resources .............................................................. 117
How to get IBM Redbooks ................................................. 118
Help from IBM ............................................................... 118
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>Cloudscape™</th>
<th>Lotus®</th>
<th>Sametime®</th>
</tr>
</thead>
<tbody>
<tr>
<td>DB2®</td>
<td>Notes®</td>
<td>WebSphere®</td>
</tr>
<tr>
<td>Domino®</td>
<td>Rational®</td>
<td>Workplace™</td>
</tr>
<tr>
<td>IBM®</td>
<td>Redbooks™</td>
<td>Workplace Managed Client™</td>
</tr>
<tr>
<td>Lotus Notes®</td>
<td>Redbooks (logo)™</td>
<td></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, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates.

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

Visual Basic, Windows, 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® Redpaper covers composite applications for desktop solutions that require multiple types of applications to run in an IBM Lotus® Expeditor V6.1 collaboration environment by providing inter-communication between components. Service-oriented architecture (SOA) composite applications deliver high levels of business services and this IBM Redpaper covers the architecture, available tools, component considerations, assembling, deploying and wiring components in Lotus Expeditor V6.1 composite applications.

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 until such time as a business deems it necessary to fully exploit the client platform, controlling access to applications based on roles assigned to users and improving the user experience by allowing composite applications’ components to easily communicate between them.

This IBM Redpaper 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 IBM Redpaper

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

Juan R. Rodriguez is a Consulting IT professional at the IBM ITSO Center, Raleigh. He has an M.S. degree in computer science from Iowa State University. He writes extensively and teaches IBM classes worldwide on 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 Coqueiro Barbosa is a Certified IT Specialist in the Lotus Software Group in IBM Brazil. He holds a master's degree in software engineering. He has over ten years of experience in object-oriented development. His areas of expertise include IBM WebSphere® Portal, WebSphere Portlet Factory, IBM 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.

Rafael Sanchez is an IT Architect with IBM Spain. He has B.S. and M.S. degrees in computer engineering, a bachelor degree in business administration, and a music teacher degree from the Royal Conservatory of Music of Madrid. He has years of experience in pervasive computing areas, including mobile telephony, embedded devices, wireless networks, and Radio Frequency Identification (RFID). He is currently doing 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 computer business working. In recent years, Robert has focused on the new, Java-based client platforms such as IBM Workplace Managed Client™, Expeditor and the upcoming Hannover technology.

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

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

Janet Dmitrovich, Charles Imperato, Rahul S. Kurane, Keith Carter
IBM Austin, Texas

Jennifer Heins, Dan Kehn
IBM Research Triangle Park, North Carolina

Niklas Heidloff
IBM Paderborn, Germany

Lubna Esmail
International Technical Support Organization, Bangalore, India
Become a published author

Join us for a two- to six-week residency program! Help write an IBM Redbook 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'll 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 papers to be as helpful as possible. Send us your comments about this Redpaper or other Redbooks™ in one of the following ways:

► Use the online Contact us review redbook form found at:
  ibm.com/redbooks
► Send your comments in an e-mail to:
  redbooks@us.ibm.com
► Mail your comments to:
  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400
Composite applications

This chapter introduces the key concepts related to composite applications. You will find 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 be tightly integrated with a WebSphere Portal server environment.

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

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 paper, 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.

1.2 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, whereas managed client composite applications are instantiated at the client side.
Server based composite application pattern

Figure 1-1 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 find out 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.
1.3 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.

### 1.4 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.
1.4.1 Overview

Portal-managed composite applications can be described and deployed on WebSphere Portal server as they are centrally managed from the portal. Figure 1-3 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.

1.4.2 Client Composite Application Infrastructure

The 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. 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 on this Lotus Expeditor feature can be found in 1.5.3, “Intercommunication between components” on page 12,
and in Chapter 4, “Component communication” on page 61, 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.

1.4.3 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.

### 1.5 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.

#### 1.5.1 Composable components in Lotus Expeditor

With Lotus Expeditor you can build a composite application from any component employing views based on any of these technologies:

- 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
1.5.2 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 on 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 3, “Managed applications” on page 33.
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 on programmatically laying out composite applications components can be found in 3.9, “Laying out applications programmatically” on page 53.

1.5.3 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 1-4.

![Diagram of Source Component and Target Component](image)

**Figure 1-4  Input and out properties**

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 *wire*.

**Note:** The concept of *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.

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 1-5 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 inter-communicate two components in Lotus Expeditor, such as broadcasting and Click-to-Action (C2A). You will find more information on these additional communication features in Chapter 4, “Component communication” on page 61.

**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 4, “Component communication” on page 61, 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.
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 chapters describes the following components:

- Local Eclipse components. This provides a short overview on the concepts of an Eclipse plug-in.
- Portlets. This introduces the issues that you have 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.
2.1 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 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 4, “Component communication” on page 61 elaborates on this particular subject in greater detail.

2.2 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
2.3 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.

2.3.1 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 2-1, the plugin.xml file must always include the extension point com.ibm.eswe.workbench.WctWebApplication.

**Example 2-1 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 2-2, the supported portlet modes are also defined here.

**Example 2-2  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>
```

### 2.4 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 2-3, the plugin.xml you must include the extension point com.ibm.eswe.workbench.WctWebApplication.

Example 2-3  Sample WctWebApplication extension point

```xml
<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"/>
    <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/or 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 2-4 shows a sample extension point for the com.ibm.pvc.webcontainer.application.

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

```xml
<extension point="com.ibm.pvc.webcontainer.application">
  <contextRoot>/com.ibm.itso.compapp.carrental.listcontracts</contextRoot>
  <contentLocation>WebContent</contentLocation>
</extension>
```
2.5 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 2-1, WSRP components in composite applications access remote portlets from a producer portal.

![Figure 2-1 Accessing remote portlets as components](image)
2.5.1 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 2-2.

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 on 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 2-5 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 2-6 on page 27.

**Example 2-5  Sample WSDL file for WSRP**

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:oasis:names:tc:wsrp:v1:wsdl">
  <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="http://<portal>/wsrp/WSRPBaseService"/>
    </wsdl:port>
    <wsdl:port binding="bind:WSRP_v1_ServiceDescription_Binding_SOAP" name="WSRPServiceDescriptionService">
      <soap:address location="http://<portal>/wsrp/WSRPServiceDescriptionService"/>
    </wsdl:port>
    <wsdl:port binding="bind:WSRP_v1_PortletManagement_Binding_SOAP" name="WSRPPortletManagementService">
      <soap:address location="http://<portal>/wsrp/WSRPPortletManagementService"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Note: This optional interface is not implemented in WebSphere Portal.
2.5.2 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 3.6.3, “Placing a WSRP portlet viewer” on page 47. As illustrated in Example 2-6, the WSRP portlet viewer instances are defined and contributed using the Portlet Viewer extension point com.ibm.rcp.portletviewer.WsrpPortlets.

Example 2-6 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>
```

2.6 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 2-3 shows a potential Hannover composite application.

![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 this writing, 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.
Chapter 3. Managed 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.

The following topics are presented in this chapter:

- 3.1, “Overview” on page 34: briefly sketches the parts and pieces of a managed application.
- 3.2, “Prerequisites for declaring composite applications” on page 35: lists the mandatory products and technologies.
- 3.3, “Access control and roles” on page 36: shows how to restrict certain capabilities for a group of people, while allowing them for another.
- 3.4, “Provisioning” on page 38: explains how to provide the downloadable code.
- 3.5, “Definition and placement” on page 39: describes the principles of assembling components together in a page.
- 3.6, “Providing components in a portal sense” on page 41: shows how to perform this using the available portal tools.
- 3.9, “Laying out applications programmatically” on page 53: provides a guide to perform this for non-connected clients.
3.1 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 3-1 shows the main building blocks that collaborate in this environment.

Figure 3-1 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
Chapter 3. Managed applications

the server, how they get distributed and what pieces are required. Figure 3-2 depicts an architectural view of a managed application.

![Figure 3-2 Architectural view of a managed application]

Figure 3-2 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

### 3.2 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 IBM Redpaper, 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.

### 3.3 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 3-2 on page 35 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 3-3 shows an example of a role definition for a sample application.

In Figure 3-3, 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.
3.4 Provisioning

Besides letting Lotus Expedit 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 Expedit, then they must be retrieved by Lotus Expedit 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 Expedit only retrieves the information on 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 3-1 shows an example of an update site tree.

Example 3-1  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 3-1, is illustrated in Example 3-2.

Example 3-2  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 3-7 on page 45.

### 3.5 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 2, “Components in composite applications” on page 17. 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.

### 3.5.1 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.
3.6 Providing components in a portal sense

This section describes how components are placed on a page when using the portal administration tool.

3.6.1 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
view, how big it should be, and so on. Figure 3-4 shows an example of a layout page containing two components.

Both portlets shown in Figure 3-4 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 3-5).

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.

---

**Figure 3-5  Rich Client Layout Administration**

At the bottom of the global options in Figure 3-5, 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 3-6 illustrates the definition of the booking portlet in the car rental sample application.

Figure 3-6 Rich client settings for an RCP component
In Figure 3-6, 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 3.6.2, “Placing portlets as components” on page 46.

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 3-7.
The available matching rules are illustrated in Table 3-1.

Table 3-1 Matching rules

<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>

3.6.2 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 3-8.

As shown in Example 3-3, the context root is located in the plugin.xml file.

Example 3-3 plugin.xml excerpt from a Client Services Portlet project

```
<extension point="com.ibm.pvc.webcontainer.application">
  <contextRoot>
```
3.6.3 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 3-9, when you initially place the portlet on a page, the unconfigured state is displayed.

![Figure 3-9  WSRP Rich Client Viewer enablement portlet](image)

As shown in Figure 3-10, activating the portlet's menu and selecting the Configure option will bring you to the first option window.

![Figure 3-10  WSDL address and security options](image)
Fill in the URL for your WebSphere Portal’s fully-qualified domain name followed by /wps/wsd1/. The remainder of the URL is at your disposal, and you can choose any arbitrary name. Once you press OK, the window shown in Figure 3-11 allows you to select the remote portlet that you want to show.

![Figure 3-11 Remote portlet selection](image)

The window shown in Figure 3-12 provides the option to specify whether a clone of the portlet is required to display the portlet.

![Figure 3-12 Selecting if a clone is needed](image)

Select OK to conclude the process to add a WSRP portlet on a composite application page.

### 3.6.4 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 3-4, the context root is found in the project’s plugin.xml file.

Example 3-4 plugin.xml excerpt of a Client Services Web Project

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

- 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.

3.7 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 3-13 shows the portlet parameter window.

![Manage Portlets](image)

*Figure 3-13  Example portlet parameters*
If required, the parameters specified in Figure 3-13 can be programmatically obtained in your plug-in code via a construct. Example 3-5 illustrates how you can do this.

**Example 3-5  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;
    }
}
```

### 3.8 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.
3.8.1 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 3-14 illustrates the option to save your application as a template.

![Figure 3-14 Saving an application as a template](image)

Select **Save Application as Template** to obtain the window shown in Figure 3-15 and enter the application template name and optional description.

![Figure 3-15 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 3-16 illustrates the window to create a new application using a template.

![Figure 3-16 Creating an application based on a template](image)

### 3.9 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 on 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 IBM Redpaper, 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.

3.9.1 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 3-17.

![Sample composite application](image)

Figure 3-17 Sample composite application

The `createInitialLayout` function for the sample application is illustrated in Example 3-6.

**Example 3-6 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 3. Managed applications  55
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 3-7.

Example 3-7  Sample car selection portlet component extension point

```xml
...<extension point="com.ibm.rcp.portletviewer.portlets">
  <portletData
    entityId="com.ibm.itso.compapp.carrental.choosecar.PortSelectCar"
    contextRoot="/com.ibm.itso.compapp.carrental.choosecar"
    portletname="PortSelectCar"
    portletwindowid="default">
  </portletData>
</extension>
...```

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 3-8 shows the extension point.

Example 3-8  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>
...```

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.

### 3.9.2 A sample remote portlet component

For the sample weather portlet component included in the sample application illustrated in this IBM Redpaper, 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 3-2.

Table 3-2  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/wsd1/&lt;your-own-unique-name&gt;.wsdl For example: <a href="http://lo-s630.ibm.com/wps/wsd1/itosweather-portlet.wsd1">http://lo-s630.ibm.com/wps/wsd1/itosweather-portlet.wsd1</a> Note: 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 LTPA Provide WSRP by a third-party producer (using the Username token)</td>
</tr>
<tr>
<td>Portlet</td>
<td>Select one of the listed portlets Note: 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 3-9 illustrates a sample configuration,

Example 3-9  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"
```
3.9.3 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 3-10.

Example 3-10  plugin.xml excerpt describing an Eclipse perspective

...  
<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>  
...  

3.9.4 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 3-11 shows an example of the extension point declaration.

Example 3-11  plugin.xml excerpt of a Client Services Web project

...  
<extension id="main_view" point="com.ibm.eswe.workbench.WctApplication">  
  <Application  
    DisplayName="%application.name"
Icon="icons/ITSO.png"
PerspectiveId="com.ibm.itso.comppapp.carrental.application.CarRentalPerspective"

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.
Component communication

This chapter describes how components can be prepared to participate in communication. We discuss how the necessary information allowing for communication 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
4.1 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 4-1 shows an excerpt of the ITSO Car Rental sample application used in this IBM Redpaper 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.

### 4.2 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 4-2 gives a graphical representation of a sample WSDL file.

![Graphical representation of a WSDL file](image)
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 4-2 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/or output messages.

The right column in Figure 4-2 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 4-1 illustrates an example of a WSDL file that is used to declare the
credit card authorization process.

Example 4-1  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>
</xsd:schema>
</types>

<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"/>
  </operation>
  <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>
</binding>
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 receiveAuth0peration 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 receiveAuth0peration. 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 4-3 illustrates an example of what the WSDL editor looks like.

![WSDL editor in Eclipse and RAD](image)

Figure 4-3 WSDL editor in Eclipse and RAD
4.3 Preparing components for inter-communication

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.

4.3.1 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 4-4 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 4-5.

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 4-6.
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 4-2.

#### Example 4-2  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 4-3, the first step in this method is to find out 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 4-3  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, the portlet 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 4-4.

Example 4-4  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 pre-defined 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 4-5.

Example 4-5  JSP page for send message portlet

```jsp
<%  
    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 4-6 shows the processing of an action in the target portlet.

Example 4-6  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 4-1 shows the relations between all parameters.

Table 4-1  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 SEND_MESSAGE_ACTION</td>
<td>MessageWireAction</td>
</tr>
<tr>
<td>SendMessage WSDL</td>
<td>&lt;binding &lt;portlet:action name</td>
<td>MessageWireAction</td>
</tr>
<tr>
<td></td>
<td>SEND_MESSAGE_ACTION</td>
<td></td>
</tr>
<tr>
<td>Plugin.xml</td>
<td>extension id=</td>
<td>targetname=</td>
</tr>
<tr>
<td></td>
<td>“com.ibm.rcp.portlet.wire”</td>
<td>“MessageWireAction”</td>
</tr>
<tr>
<td>ReceiveMessage WSDL</td>
<td>&lt;binding &lt;portlet:action name</td>
<td>MessageWireAction</td>
</tr>
<tr>
<td></td>
<td>SEND_MESSAGE_ACTION</td>
<td></td>
</tr>
<tr>
<td>ReceiveMessage Portlet</td>
<td>public String SEND_MESSAGE_ACTION</td>
<td>MessageWireAction</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 4-7.

**Example 4-7  WSDL location in portlet descriptor (portlet.xml) file**

```
<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 WSD file for the source portlet is provided for you as additional material in this IBM Redpaper. For details see Appendix A, “Additional material” on page 115.

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

1) From the Project Explorer view, right-click on the DemoWire/WebContent folder and select New → Folder.

2) Enter wsdl in the folder name field and click Finish.

3) Right-click on the wsdl folder just created and choose File → New → Other → WebService → WSDL and fill in the wsdl properties as shown in Figure 4-7.

![Figure 4-7 Using the WSDL editor](image)

Example 4-8 illustrates a sample WSDL for a source portlet when publishing a message to the property broker.

**Example 4-8  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">

<types>
    <xsd:schema targetNamespace="http://www.ibm.com/wps/c2a">
        <xsd:complexType name="WireStringType">
            <xsd:restriction base="xsd:string">
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:schema>
</types>
```
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 4-9 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 4-9  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"
```
```xml
<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>
```
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 managed application environment, wires can also be defined and contributed to the Lotus Expeditor platform when using the
portal-managed environment. For details see 4.4.2, “Wires declared through WebSphere Portal” on page 89.

The sample scenario shown in this chapter relates to the non-portal managed 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 4-8.

![Figure 4-8 Plug-in extensions](image-url)
3. Select Add.

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.

Example 4-10   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=""
        sourceentityid="/DemoWire/SendMessage/default"
        sourcename="wire_text"
        sourceparam=""
        targetentityid="/DemoWire/ReceiveMessage/default"
        targetname="MessageWireAction"
        targetparam="wire_text"
        title=""
        type="PROPERTY_TO_ACTION"/>

</extension>
```
4.3.2 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 a 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 4-11 illustrates how a property broker change event in an Eclipse IHandler action is processed.

Example 4-11 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 4-12 illustrates the broker's event interface in an SWT IAction.
Example 4-12   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 4.2, “Declaring data items and actions” on page 63 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 4-13   plugin.xml excerpt showing a property broker definition

```xml
...<extension
  point="com.ibm.rcp.propertybroker.PropertyBrokerDefinitions">
    <handler class="com.ibm.itso.comapp.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 section 4.4, “Declaring wires” on page 86, 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 4-14 illustrates how to create a property and assign a value to it.

**Example 4-14  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 later in 4.4, “Declaring wires”. This is the safest way to communicate. Firstly, it makes sure that no other component receives your data accidentally, and in worst cases this could even pose a security hole. Secondly, it obeys the wishes 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 bare 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 3.9, “Laying out applications programmatically” on page 53, 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 4-15 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 4-15  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 4-16 shows potential coding for the $SWTHelper.changedProperties$ method.

**Example 4-16  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.

### 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.

**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 4-17 shows you how to do that.

Example 4-17  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:** 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 4-18 illustrates a Right-Mouse-Button-Context-Menu implementation for an SWT table viewer control.

Example 4-18  Right mouse button context menu implementation

```java
public void mouseUp(MouseEvent arg0) {
    if (arg0.button == 3){
        String selection = viewer.getSelection().toString();
```
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.

### 4.4 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 81, 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 4.2, “Declaring data items and actions” on page 63, 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.

### 4.4.1 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 81, 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 4-2.

<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 API's with this string. Otherwise, the owner ID will be populated by the property broker.</td>
</tr>
<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 value of your portlet:param tag in the output section.</td>
</tr>
<tr>
<td>sourceentityid</td>
<td>This is the source's entity ID.</td>
</tr>
<tr>
<td>sourceparam</td>
<td>The source parameter that will be included with the wire information when 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 ID.</td>
</tr>
<tr>
<td>targetparam</td>
<td>The target parameter that will be included with the wire information when sent to the Action. Portal will put the target's property name into this parameter.</td>
</tr>
<tr>
<td>enable</td>
<td>This will have the wire automatically enabled when it is registered.</td>
</tr>
</tbody>
</table>
A practical example of a plugin.xml-based wire definition is described in Example 4-19. 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 4-19  Example plugin.xml based wire

```
<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"
        targetparam="CC Credit Card Number"
        title="Publish the credit card number"
        type="PROPERTY_TO_ACTION" />
</extension>
```

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.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>uid</td>
<td>(optional) A unique ID for this specific wire. If not supplied, a unique ID 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.
4.4.2 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 4-20 on page 90 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 3.5.1, “Portlet instances versus portlet clones” on page 40. This is going to change if WSDL files are involved. As you can not 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 4-10 shows you what your project will look like.

![Figure 4-10](image)

*Figure 4-10  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 4-11 shows what it would look like.

![Portlet Deployment Descriptor editor](image)

Figure 4-11 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 4-20, to your portlet.xml.

**Example 4-20 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>
...
```

**Note:** Add these lines inside the tag that declares your portlet.
The ITSO Car Rental sample composite application

This chapter contains step by step instructions on how the ITSO Car Rental sample application was developed. Firstly, it gives you an introduction to the basic architecture of the application and eventually leads to the sample steps on how to deploy the ready application to a WebSphere Portal for provisioning to Lotus Expeditor client desktops.
5.1 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 IBM Redpaper, 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 paper. 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 enables 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 5-1.

![Diagram of the ITSO Car Rental for the representative role](image)

**Figure 5-1** 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 5-2.

![Diagram of the ITSO Car Rental for the supervisor role](image)

5.2 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.
5.2.1 Customer model

This section describes the customer table and data bean.

Customer record

The statement shown in Example 5-1 is used to create the customer table.

Example 5-1 SQL-Statement to create the customer table

```
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 5-1.

Table 5-1 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>
</tbody>
</table>
5.2.2 Car model

This section describes the car table and data bean.

**Car record**
The statement shown in Example 5-2 is used to create the car table.

*Example 5-2 SQL-Statement to create the car table*

```
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 5-2.

<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>
<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>

### 5.2.3 Contract model

This section describes the contract table and data bean.

**Contract record**

The statement shown in Example 5-3 is used to create the contract table.

*Example 5-3  SQL-Statement to create the contracts table*

```sql
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.comapp.carrental.database.Contract` can be used to represent such a record. It provides getters and setters for the fields shown in Table 5-3.

<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>CUSTID</td>
<td>String custid</td>
<td>getCustid</td>
<td>setCustid</td>
</tr>
<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>

**5.2.4 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

▶ getContractsByCustID
Retrieves an array of contract records that satisfy the provided search ID,
which can be a wildcard in the SQL notation. For example, 12% to find all
contracts starting with 12. The method syntax is:

public static Contract[] getContractsByCustID(java.lang.String id)
– Parameters
   ID or wildcard ID
– Returns
   An array of all contracts found or an array of zero size if either no contracts
   were found or an exception occurred

▶ writeNewContract
Writes a new contract record to the contract table in the database. The
method syntax is:

public static boolean writeNewContract(Contract contract)
– Parameters
   A ready and populated contract bean to write the record
– Returns
   True, if the record was successfully written to the table, false otherwise

5.3 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 5-6 on page 101.
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 5-4 shows the added start() method.

Example 5-4   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);
}
...

ImageData
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 5-5.

Example 5-5   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 5-3 illustrates the three sections or parts in the BookingComposite 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 5-6  Booking component’s MANIFEST.MF file*

```
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-Activator:
com.ibm.itso.compapp.carrental.booking.BookingActivator
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 5-7 shows the content of the plugin.xml file.

Example 5-7  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>
```

5.4 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.

**Figure 5-4   Car Selection portlet**

![Car Selection portlet](image)

**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 5-8.

**Example 5-8   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;
```
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 require 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 5-9 shows the doView() method in the sample portlet component.

Example 5-9  Method doView()

```java
public void doView(RenderRequest request, RenderResponse response) throws PortletException, IOException {
    response.setContentType(request.getResponseContentType());
    request.setAttribute("carList", carList);
    request.setAttribute("wirePreferredCarFromBooking", wirePreferredCarFromBooking);
    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 list 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 5-10 shows the `processAction()` method.

**Example 5-10  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 5-11.

**Example 5-11  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 5-12.

**Example 5-12  JSP Form**

```jsp
<form method="POST" name="frmCar" action="<%= actionUrl.toString() %>>">
<
    <div id="Selection">Select your car size: <select name="selTypeCar">
        <option value="C"<%if(wirePreferedCarFromBooking.equals("C")) out.println("selected"); %>>Compact</option>
        <option value="M"<%if(wirePreferedCarFromBooking.equals("M")) out.println("selected"); %>>Medium</option>
        <option value="F"<%if(wirePreferedCarFromBooking.equals("F")) out.println("selected"); %>>Full</option>
    </select>
    <input
```
name="<%=com.ibm.itso.compapp.carrental.choosecar.PortSelectCar.FORM Submit%>">
    type="submit" value="OK">
</div>
<input type="hidden" name="selCarH">
<table>
    <tbody>
        <tr>
            <th>Photo</th>
            <th>Manufactor</th>
            <th>Model</th>
            <th>Rate</th>
            <th>Color</th>
            <th>Shift</th>
        </tr>
        <% int numberCars = carList.length;
            for (int x = 0; x < numberCars; x++) {
            %>
            <tr>
                <td><a href="javascript:sendCar(<%= x %>)"><img src='"<%=renderResponse.encodeURL(renderRequest.getContextPath() + "/images/" + carList[x].getMake() + ".jpg") %>' alt=""></a></td>
                <td><%=carList[x].getMake()%></td>
                <td><%=carList[x].getModel()%></td>
                <td><%=carList[x].getRate()%></td>
                <td><%=carList[x].getColor()%></td>
                <td><%=carList[x].getShift()%></td>
            </tr>
            <% }
        %>
    </tbody>
</table>
</form>
5.5 The credit card component

In this section the credit card component is described.

5.6 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 5-13 shows the model portion of the component.

Example 5-13 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;
while (rs.next() && a < records) {
    contracts[a] = new Contract(rs);
}
if (conn != null) conn.close();

} catch (SQLException e) {
    e.printStackTrace();
}
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 5-14 shows the doGet() method.

Example 5-14  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 5-15 illustrates the doPost() method.

**Example 5-15  doPost method**

```java
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 5-16  JSP Page**

```html
<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" maxlen="10"></td>
    </tr>
    <tr>
      <td width="150">
        <input type="submit" name="cmdOK" value="OK">
        <input type="reset" value="Clear">
      </td>
    </tr>
  </tbody>
</table>
```

110 Building Composite Applications in Lotus Expeditor V6.1
Chapter 5. The ITSO Car Rental sample composite application

5.7 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.

### 5.7.1 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 5-6 illustrates the sample portlet.

![Weather portlet](image)

**Figure 5-6  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 5-7.

![Figure 5-7  Selecting a portlet to be provided](image1)

4. Click **OK** to provide the portlet. See Figure 5-8.

![Figure 5-8  Confirming the operation](image2)

5. A message appears indicating that the portlet has been successfully provided. As shown in Figure 5-9, a check mark appears in the Provided column of the portlet.

![Figure 5-9  Produced portlet for remote access](image3)
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 5-10.

![Figure 5-10  Withdrawing a produced portlet](image)

3. Click **Yes** to withdraw the portlet. A message appears indicating that the portlet has been successfully withdrawn.
Additional material

This IBM Redpaper refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this IBM Redpaper is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

ftp://www.redbooks.ibm.com/redbooks/REDP4241

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select Additional materials and open the directory that corresponds with the IBM Redbook form number, REDP4241.

Using the Web material

The additional Web material that accompanies this IBM Redpaper includes the following files:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>redp4241.zip</td>
<td>Sample ITSO Car Rentals composite application</td>
</tr>
</tbody>
</table>
System requirements for downloading the Web material

The following system configuration is recommended:

- **Hard disk space:** 80GB minimum
- **Operating System:** Windows® or Linux®
- **Processor:** 2HZ 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 IBM Redpaper.

IBM Redbooks

For information about ordering these publications, see “How to get IBM Redbooks” on page 118. Note that some of the documents referenced here may be available in softcopy only.

- Patterns: SOA Client - Access Integration Solutions, SG24-6775
- IBM WebSphere Everyplace Deployment V6 Handbook for Developers and Administrators Volume I: Installation and Administration, SG24-7141
- 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:

- Lotus Expeditor product documentation

Online resources

These Web sites are also relevant as further information sources:

- Eclipse
  http://www.eclipse.org/
- Lotus Expeditor homepage
  http://www.ibm.com/software/sw-lotus/lotus_expeditor
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
Building Composite Applications in Lotus Expeditor V6.1

This IBM Redpaper covers composite applications for desktop solutions that require multiple types of applications to run in a Lotus Expeditor V6.1 collaboration environment by providing inter-communication between components. SOA composite applications deliver high levels of business services and this IBM Redpaper covers the architecture, available tools, component considerations, assembling, deploying and wiring components in Lotus Expeditor V6.1 composite applications.

Lotus Expeditor is a client platform for end-to-end smart client application solutions. Expeditor provides services to install and manage these applications and allow 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 until such time as a business deems it necessary to fully exploit the client platform, controlling access to applications based on roles assigned to users and improving the user experience by allowing users to easily switch among multiple applications.