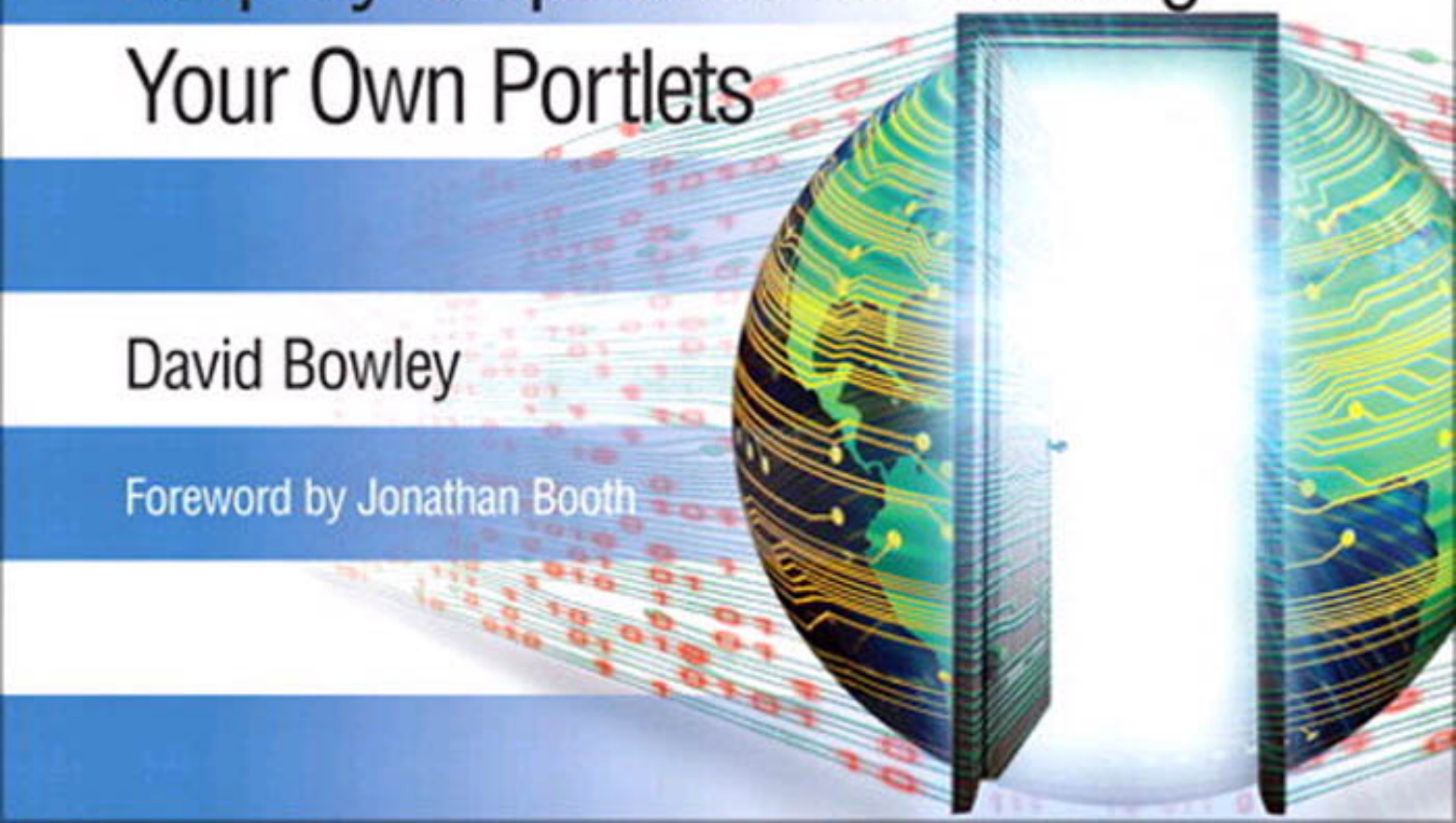
IBM PRESS

Rapid Portlet Development with WebSphere Portlet Factory

Step-by-Step Guide for Building Your Own Portlets



David Bowley

Foreword by Jonathan Booth

IBM Press Program Managers: Tara Woodman, Ellice Uffer
Cover Design: IBM Corporation

Associate Publisher: Greg Wiegand
Marketing Manager: Kourtnaye Sturgeon
Publicist: Heather Fox
Acquisitions Editor: Katherine Bull
Development Editor: Kevin Howard
Managing Editor: Kristy Hart
Designer: Alan Clements
Senior Project Editor: Lori Lyons
Copy Editor: Deadline Driven Publishing
Indexer: WordWise Publishing Services
Compositor: Nonie Ratcliff
Proofreader: Water Crest Publishing
Manufacturing Buyer: Dan Uhrig

Published by Pearson plc

Publishing as IBM Press

IBM Press offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales, which may include electronic versions and/or custom covers and content particular to your business, training goals, marketing focus, and branding interests. For more information, please contact:

U. S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside the U.S., please contact:

International Sales
international@pearsoned.com

Library of Congress Cataloging-in-Publication Data

Bowley, David.
Rapid portlet development with WebSphere portlet factory : step-by-step guide for building your own portlets / David Bowley.
p. cm.
Includes index.
ISBN 0-13-713446-0 (hardback : alk. paper) 1. Web portals—Computer programs. 2. User interfaces (Computer systems) Computer programs. 3. Web site development. 4. WebSphere. I. Title.
TK5105.8885.W43B69 2008
006.7’6—dc22
2008029014

ISBN-13: 978-0-13-713446-5
ISBN-10: 0-13-713446-0

Text printed in the United States on recycled paper at R.R. Donnelley in Crawfordsville, Indiana.
First printing September 2008

Foreword

Building good software applications is hard. Improvements in languages, frameworks, and tools do make things easier, and there are more of these improvements each year.

But at the same time, the technology landscape that developers live in keeps changing and getting more complex. Just when you get productive with one set of tools and technology, there's something new that you have to adapt to or integrate with. And there's a perpetual demand for "more software quicker"—organizations can never get all the software they want as soon as they want it.

WebSphere® Portlet Factory was created to apply concepts of *software automation* to help address this ongoing problem of software development complexity. This software automation moves the developer up a level, above the level of individual code artifacts. Instead of directly manipulating elements such as JSP, Java™, JavaScript, and XML files, the developer interacts with *builders* in a model, and the builders then generate all the necessary code artifacts in response to the developer's high-level instructions.

You can think of builders as encapsulations of software *features* or *design patterns*. Each builder implements one feature of an application, controlled by instructions provided by the developer in a wizard-like user interface. An application is built by successively adding and modifying features (builders) until the application works as intended. The net effect for developers is that they can rapidly develop complex applications without having to learn (and remember) all the underlying technology.

In the past several years working with this technology, we've found that developers can consistently get big productivity gains from this software automation. We've seen the technology adopted by an ever-increasing customer base, first at Bowstreet (where the software was initially developed), and now at IBM, which acquired Bowstreet in late 2005. At IBM, the technology has

also been adopted by a number of other product groups that build products on top of Portlet Factory technology and take advantage of its software automation. For example, the Lotus® ActiveInsight Dashboard Framework is built on Portlet Factory and provides a set of builders that implement high-level design patterns tailored for dashboard-style applications.

We've also found that automation makes it possible to quickly add support for new technology, such as integrating new back-end services or generating new user interface technologies. One example is support for Ajax (Asynchronous Java and XML) user interfaces. Implementing an Ajax user interface through hand-coding is quite complex and involves coordinated client-side code (JavaScript) and server-side code. Using builder technology, a small team with Ajax expertise was able to capture their expertise in a set of builders that automate common Ajax patterns and generate the necessary client and server code. Once the builders were created, those Ajax patterns became easily accessible to any developer using Portlet Factory.

In this book, David Bowley gives a clear "soup-to-nuts" guide to building applications with Portlet Factory, from creating your first project, to back-end integration, to user interface and Ajax techniques. Each chapter tackles one aspect of application development, and for each task David shows you which builders you need and how to use them. In his examples, I think you'll see that David has found just the right level of complexity—the examples are simple enough to easily understand, but not unrealistically simple or trivial.

Portlet Factory development—using builders and models instead of working directly with code—represents a different development paradigm than with other tools. I hope you find as much value in this automation paradigm as we have. You can use this book as your guide as you learn your way around Portlet Factory and get comfortable with this way of working.

Jonathan Booth
Senior Architect
WebSphere Portlet Factory
IBM

Preface

Portlet development can often be arduous and complicated; indeed, the word "rapid" is not normally associated with building portlets. IBM's award-winning[1] WebSphere Portlet Factory (WPF), however, provides developers with a wizard-based development environment that greatly expedites the process of building, testing, and deploying portlets. WPF shields developers from much of the complexity of traditional portlet development, and portlets built using WPF often require little or no coding—enlarging the potential pool of people who are able to build portlet applications. Having said this, WPF developers also have the full power of Java 2 Enterprise Edition (J2EE) available to them should they choose to use it, making WPF a flexible (and powerful) development tool.

This book is about how to use WPF to rapidly build portlets. No previous development experience is required to understand this book, and anyone with a remote interest in portlet development should find something of interest here. The book is structured to facilitate rapid portlet development: It is a collection of independent chapters, each walking through the process of creating a portlet while focusing on a particular aspect of WPF. Due to the independent nature of the chapters (and the nature of portlet development using WPF), you can skip to the chapters that interest you without needing to read chapter after chapter of abstract theory and/or background information beforehand. For example, if you want to learn how to build a portlet that displays a graphical chart, skip to Chapter 10, "Using Charts in Portlets;" if you want to find out how to work Ajax into your portlets, skip to Chapter 13, "Using Ajax and Dojo." If you are completely new to WPF (or portals and portlets) and are looking for some basic information to get you started in WPF, Chapter 1, "Introduction to WebSphere Portlet Factory," provides an overview of

[1] WebSphere Portlet Factory won the 2006 JavaPro readers' choice award for "Best Java Enterprise Portal Technology" (www.fawcette.com/javapro/).

portal terminology, WPF architecture, and the WPF Designer interface. Chapter 1 also walks you through the process of creating, testing, and deploying a simple Hello World! portlet. Other introductory information is available in Appendix A, which contains some useful information for setting up your WPF development environment, and there is a glossary at the back of the book that defines common WPF terms.

This book also contains useful tidbits that I have picked up during my time with WPF—the sort of things developers need to know but normally wouldn't without a great deal of experimentation and frustration. These snippets of information are highlighted as follows:

> **TIP**
>
> Tips contain useful information that can be employed in WPF, usually with the purpose of expediting portlet development. Tips are useful, but they are not critically important and can be skipped if desired.

> **WARNING**
>
> Warnings are important points that usually obviate a common sticking point, and heeding them may spare you future frustration.

Due to the width of the book's printed page, some lines of code might be too long to fit on only one line. If a line of code wraps to another line(s), and it's not obvious it's part of the preceding line, we have inserted a code continuation character ([ccc]) at the beginning of the runover line to indicate that the code is part of the line preceding it. (You should type lines of code that have this character as one long line without breaking it.) For example,

```
WebAppAccess remoteWebAppAccess =
➥webAppAccess.getModelInstance("modelPath/ModelName", "",
➥false);
```

All the examples discussed in this book are available for download from ibmpressbooks.com/title/9780137134465. More advanced readers can import what they want from these examples directly into their projects, without necessarily consulting the book itself—although you are encouraged to follow along with the examples to increase your understanding of WPF. By walking through each example, you will learn how to build portlets in WPF by actually *building* them, and not just reading about it; so, by the end of each chapter, you should have a practical understanding of how to work the discussed features into your own portlets.

Although this book does discuss the theory of WPF portlet development, this information is discussed in the context of the practical examples in the book, which gives you a more concrete

understanding of how the abstract side of portlet development is applied. Readers unconcerned with what is going on under the covers can skip the theory sections without adversely affecting their portlets. Indeed, one of the advantages of using WPF is that you don't need to learn vast amounts of theory to begin development—you can start building portlets right away. The focus of this book, then, is on the practical side of portlet development, with an emphasis on rapidly building portlets with certain functionality—this is not intended as a book of theory. Similarly, given its short length, this book is not intended to cover every aspect of portlet development—only those areas that are deemed most useful to portlet developers (and WPF developers, in particular).

I hope you find this book useful and enjoyable to read; I certainly enjoyed writing it. At the least, I would like this book to go some way toward expediting your portlet development process and increasing your understanding of WPF. If you have any comments about the content or structure of the book, feel free to drop me a line at dave.bowley@gmail.com

CHAPTER 7

Communicating Between Portlets

The ability to send information from one portlet to another adds considerable flexibility to your portlet applications, in both the way they are designed and in the way they are used. This chapter outlines the numerous ways that you can implement inter-portlet communication using WebSphere Portlet Factory (WPF). By the end of this chapter, you will understand the strengths and weaknesses of each approach and will have built a library loans portlet application to demonstrate them.

Each of the models in this chapter is available for download from ibmpressbooks.com/title/9780137134465 under the Chapter 7 folder (instructions for copying these files into your project are included in a readme.txt file in the same folder); however, to increase your understanding of the topics discussed, it is recommended that you create the models yourself by following through the example in this chapter.

Note that the example in this chapter does not go into detail about the service/provider consumer pattern; for more information on this pattern and on services in general, refer to Chapter 2, "Providing and Consuming Services."

The following topics are covered in this chapter:

- The benefits of inter-portlet communication
- The WebSphere Property Broker
- Creating a project
- Creating a service provider
- Creating a list portlet
- Creating a detail portlet

- Configuring the WebSphere Property Broker
- Alternative communication methods
- When to use inter-portlet communication

The Benefits of Inter-Portlet Communication

Working inter-portlet communication into your applications has several benefits from both a design perspective and a usability perspective. Some of the key benefits are listed below.

Extensibility
Inter-portlet communication makes your applications easier to extend and plug into other applications. For example, a portlet that displays a list of loans for a library database might allow users to click on a particular loan, and then send a loan ID off to other portlets to perform an appropriate action. One portlet might be used to display details for the loan, and another portlet might display a chart of overdue loans for the borrower. In this example, you could develop additional portlets that interact with either portlet, without having access to the source code for the loans portlets.

Maintainability
Inter-communicating portlets are easier to maintain because you can develop or modify a portlet without making changes to others, assuming that you do not change the nature of the content that is communicated. For example, you can change the user interface and services in the loans list portlet without worrying about affecting the loan detail portlet.

Usability
Because the interface for applications that use inter-portlet communication can be spread across several portlets, inter-portlet communication can be beneficial for users. Depending on their access rights, users can drag and drop these portlets around the portal page to suit their personal preferences, and they can add and remove portlets to further customize the interface to their application.

The example application discussed in this chapter demonstrates inter-portlet communication for a simple library system between a portlet listing library loans, and another portlet displaying details on an item in the list. The application utilizes three models: a list model, a detail model, and a service provider model. The list model is surfaced to a portal server as a portlet and displays the list of loans. The detail model is also surfaced as a portlet and displays details for a loan selected in the list model. Finally, the service provider provides the list and detail data for the list and detail models, which is stored in an XML document.

The WebSphere Property Broker

JSR-168 portlets do not natively support inter-portlet communication, but portlets can still communicate with each other in WebSphere Portal Server using a mechanism called the WebSphere

Property Broker. The Property Broker is the preferred method of inter-portlet communication in WebSphere Portal, and it is the first communication method demonstrated in this chapter.

> **TIP**
>
> JSR-168 is a standard set of Java APIs for developing portlets. You can and should specify your WPF portlets as JSR-168 by selecting Java Standard for the Portlet API setting in your portal server deployment configuration. The Property Broker receives properties from portlets, such as a loan ID or success flag, and then publishes these properties for the rest of the portal to use. Target portlets can then access these properties and define actions to respond to property changes. After you have deployed your portlets in WPF, you then use the portal administration interface to set up a link between a property in a source portlet and a property in a target portlet. This process is known as wiring two portlets together.

The WebSphere Property Broker can facilitate communication between portlets in different applications, and also between portlets that were built using different development tools. For example, a WPF portlet can send information to a standard Java JSR-168 portlet, and vice-versa. This makes it easy to maintain and extend your applications, and other developers can write portlets that communicate with your portlets without needing access to the source code.

The first few sections in this chapter focus on using the WebSphere Property Broker to set up inter-portlet communication between a library list portlet and a library detail portlet. The "Alternative Communication Methods" section then outlines some potential alternatives for communicating between portlets.

Creating a Service Provider

To provide data to your list and detail portlets, you should create a service provider model. This model will contain operations to retrieve a list of loans and retrieve details on a particular loan, which is consumed by the list and detail models. The service provider is the singular access point for loans information in your application, and it makes your application easier to maintain.

Before you can begin the steps in this section, you need a WPF project, which houses the service provider model and the other models in this chapter. If you have a project from a previous chapter, you can use that. If you don't, you should create a new WPF project. For more information on creating projects, see Chapter 1, "Introduction to WebSphere Portlet Factory." The project is published as a WAR file and deployed to a portal server, and you should also deploy the application to a local application server for testing. You can use the portal server if it is running on your local machine. If not, it is recommended that you use the IBM WebSphere Application Server Community Edition server (WAS CE) that comes with WPF.

After you have a project set up, you need to add the service provider model. The service provider uses the following builders to provide its functionality:

- Service Definition
- Variable
- Simple Schema Generator
- Action List
- Method
- Service Operation (x2)

Note that although the order of builder calls is usually not important, some of the builder calls in this chapter need to be in a specific order. This is because some of the builder calls require certain things to be built by the time their build methods run when the WebApp is regenerated. In this example, the Variable builder call must precede the Simple Schema Generator builder call to create a schema based on the variable built by the Variable builder call, and the Simple Schema Generator must precede the Service Operations because the Service Operations require the schema to use for their results. Also, the Action List must precede the first Service Operation, and the Method must precede the second Service Operation, or you will get errors indicating that these resources are unavailable.

Creating a Model

Create a new model called `loansService` in your project under the folder WEB-INF/models/chapter09. Because you will store your data in an XML variable, rather than get it from a database, you need to create the service provider manually. To do this, the model should be based on the Empty template, which creates a model with no builder calls in it. For more information on creating models, see the example in Chapter 1.

Defining the Service

Add a Service Definition builder call to the model by selecting Service Definition from the Builder Palette. You can open the Builder Palette by clicking the  icon in the Outline view. Then, press OK. Name the builder call `loansService` and expand the Testing Support section at the bottom of the builder call, and then enable the box for Add Testing Support. WPF now automatically generates test pages for the operations in your service every time the model is regenerated. Save the builder call when you are finished.

Adding Loan Data

The next step is to add your sample data for the service provider. Add a Variable builder call to the model and name it `loans`. Change the Type input to XML and enter in the following XML into the Initial Value input:

```
<Loans xmlns="http://com.ibm.CooperativeExample">
   <Loan>
      <LoanID>00001</LoanID>
```

```
        <Item>The Gullible Traveler</Item>
        <LoanedTo>Malcolm Johnson</LoanedTo>
        <DueDate>11/05/2007</DueDate>
    </Loan>

    <Loan>
        <LoanID>00002</LoanID>
        <Item>A Traveling Aunt</Item>
        <LoanedTo>Samuel Russell</LoanedTo>
        <DueDate>11/11/2007</DueDate>
    </Loan>

    <Loan>
        <LoanID>00003</LoanID>
        <Item>An Interminable Journey</Item>
        <LoanedTo>Joseph Jones</LoanedTo>
        <DueDate>12/22/2007</DueDate>
    </Loan>
</Loans>
```

Save the builder call when you are finished. Next, add a Simple Schema Generator builder call to the model. This builder call is used to automatically generate a schema based on the Variable builder call, which is then used to define the structure of the result sets for the Service Operation builder calls. Name the builder call `loansSchema`, and for the Sample Data input, select 'loans' from the drop-down list. This is the variable you created earlier. Save the builder call when you are finished.

Adding an Action to Retrieve a List of Loans

Now, add an Action List builder call to the model. The Action List defines the action to be called when the retrieveLoansList service operation is consumed, which you create in a later step. Name the Action List `getLoans` and change the return type to IXml. IXml is a WPF interface used to access and manipulate XML. For more information on IXml, see Chapter 9, "Using Web Services and Manipulating XML."

Open the Select Action dialog for the first row in the Actions input, which you do by clicking on the ellipsis button on the first row of the Actions input. Select Special, Return, then select the loans Variable from under the Variables section and press OK. The action should appear in the Action List as follows:

```
Return!${Variables/loans/Loans}
```

When the Action List is called, this action returns the entire Loans variable created by the Variable builder. Save the builder call when you are finished.

Specifying the getLoansList Operation

Next, you create the first operation for the loansService service. This operation returns a list of loans (by calling the getLoansList Action List). Add a Service Operation builder call to the model and make sure the Service Operation Properties section is expanded. Select loansService for the Data Service input, which links the operation to the loansService service created by the Service Definition builder call.

The Operation Name input defines how the operation is referred to by consumers. Change this input to `getLoansList`—there is no problem with this being the same name as your Action List. The Action To Call input defines what happens when the operation is consumed. This input should be changed to point to your Action List (getLoansList). Note that the Operation Description input is left blank, because it appears only to service consumers when the service provider is defined as a Web service.

Make sure 'No inputs' is selected for the Input Structure Handling input in the Operation Inputs section of the builder call. This specifies that the operation has no inputs—consumers have only to specify that they want to consume the operation to retrieve the loans list.

Expand the Operation Results section of the builder call and select Specify Result Schema for the Result Structure Handling input. This enables you to manually specify the structure of the results to be returned from the operation. For the Result Field Mapping input, select the topmost element in the Loans schema (Loans). The Loans element contains a list of Loan elements and is returned from the getLoansList Action List. The input should read loansSchema/Loans when you are finished. Make sure the Result Field Mapping input is set to Automatic, which automatically maps results from the called action to the result returned from the service operation.

Save the builder call when you are finished.

Adding a Method to Retrieve a Specific Loan

Add a Method builder call to the model and name it `getLoanFromID`. This method returns a particular loan based on a loan ID argument and is called whenever the getLoanDetail operation is consumed. A Method builder call is used, rather than an Action List, because some IXml manipulation is required to link loanIDs to loans.

Expand the Arguments section of the builder call and add a single argument called loanID of type String. This argument corresponds to a particular loan, and you access it from the method defined in the Method Body Input.

Change the Return Type of the builder call to IXml, and then enter in the following code into the Method Body input:

```
{
  //cycle through loans variable to get each loan
  IXml loans = webAppAccess.getVariables().getXml("loans");
  for (IXml loan = loans.getFirstChildElement(); loan !=null;
  loan = loan.getNextSiblingElement())
  {
```

```
        //look for match on loanID element
        if ( loan.getText("Loan/LoanID").equals(loanID) )

          //match found, return the current loan
         return loan;
      }

      //no match found, return empty loan
      return XmlUtil.create("Loan");
    }
```

This code cycles through each of the loan elements in the loans list, and checks to see whether the loan ID of the element matches the loan ID passed in to the method. For more information on the use of IXml, see Chapter 9. Save the builder call when you are finished.

Specifying the getLoanDetail Operation

The final builder call for the service provider is another Service Operation builder call. This builder call defines a second operation for the loansService service, which displays details for a particular loan (by calling the getLoanFromID Method). Add a Service Operation builder call to the model and point the Data Service input to loansService. Change the Operation Name to `getLoanDetail`, and the Action To Call to `getLoanFromID`. This creates a new operation on the loansService service called getLoanDetail, which runs the getLoanFromID method when consumed.

In the Operation Inputs section of the builder call, select 'Use structure from called action' for the Input Structure Handling input and make sure the Input Field Mapping input is set to Automatic. This specifies that the inputs to the operation are defined by the getLoanFromID Method, and should be automatically mapped to the inputs in getLoanFromID.

Fill out the Operation Results section of the builder call the same as the Operation Results section in the previous Service Operation builder call, which specifies that the structure of the results to be returned from the operation is defined by the Loan element in the Loans schema. Save the builder call when you are finished.

You have now finished building your service provider. The next section discusses how you can test your service provider before moving on to create the list portlet and detail portlet.

Testing the Service Provider

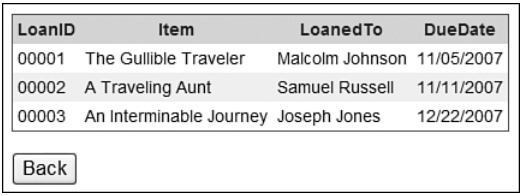
Because you enabled test support for your service, WPF automatically generates test pages for your service provider, which you can disable from the Testing Support section of the Service Definition builder call. This enables you to test the service provider directly from the IDE. To test the service, run the loansService model from the WPF Designer by clicking the  icon on the toolbar. This

runs the currently active model (for instance, loansService) with the last run configuration you used. If you have not set up a run configuration before, you are prompted to do so—create a new configuration under the WebSphere Portlet Factory Model category. If you want more information on setting up a run configuration, see the "Testing the Application" section in Chapter 1.

After you run loansService, you should see the index test page in your default Web browser, as shown in Figure 7.1. This screen lists all the operations available on the loansService service.

Figure 7.1 Index test page from the loansService model.

To test the getLoansList operation, click the getLoansList link. You should see a list of loans, as shown in Figure 7.2. If not, check that you don't have any errors showing in the Problems view in the IDE and that you have completed all the steps in the previous section correctly. Press Back to return to the index page.



LoanID	Item	LoanedTo	DueDate
00001	The Gullible Traveler	Malcolm Johnson	11/05/2007
00002	A Traveling Aunt	Samuel Russell	11/11/2007
00003	An Interminable Journey	Joseph Jones	12/22/2007

Back

Figure 7.2 Testing the getLoansList operation.

To test the getLoanDetail operation, click the getLoanDetail link. The screen that follows asks you to enter in an ID of a loan you would like to retrieve details for. Enter in `00001` as the ID and press the Submit Query button. You should see some details for the loan that has an ID of 00001 (that is, The Gullible Traveler), as shown in Figure 7.3. Press Back when you are finished.

Close the loansService model when you've finished testing it.

You now have a service provider called loansService, which provides a list of loans and details on individual loans. The next few sections walk through the process of creating consumers for the loansService service in the form of a list portlet and detail portlet.

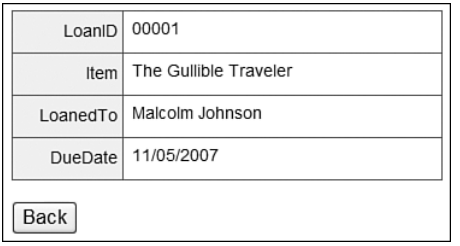


Figure 7.3 Testing the getLoanDetail operation.

Creating a List Portlet

This section describes how to add another model to your project, which is surfaced to a portal server as a portlet. The portlet consumes the service created in the previous section, and displays a list of loans to the user. When a loan in the list is clicked, the ID of the loan is sent to the WebSphere Property Broker, which can then be read by other portlets. A screenshot of both the list and detail portlets is shown later in Figure 7.15.

The model uses the following builders to provide its functionality:

- Portlet Adapter
- Service Consumer
- View & Form
- Cooperative Portlet Source

Creating a Model

To build the list portlet, select File, New, WebSphere Portlet Factory Model from the File menu to bring up the WebSphere Portlet Factory Model dialog. On the next screen, select your WPF project and press Next. The next screen lists different types of models WPF can create for you automatically. You can create service consumers in WPF in two ways: the first is to let WPF create one for you, and the second is to create the service consumer manually. The builders you use to create the list portlet in this section are fairly standard, so you can opt for WPF to create them for you. Select List and Detail Service Consumer from the Service Consumers category and press Next.

Specifying the Service

The next screen asks you define a portlet name and service provider model for the consumer. The portlet name is also used as a prefix when the wizard names some of the builder calls in your model. Name the portlet `loans` and select chapter07/loansService as the service provider model. Click Next to continue.

Specifying List and Detail Operations

You now need to specify an operation to retrieve view data—the list portion of your portlet. Select getLoansList from the drop down and press the Next button to edit the detail portion of the portlet. Although the list portlet in this chapter is intended to be used as a list portlet only, defining the detail portion gives the loansList model the flexibility to be run as a list and detail portlet together if a separate detail portlet is unavailable. Fill out the settings on this screen, as shown in Figure 7.4. This converts the loanID column values into clickable links that run the getLoanDetail operation when clicked. Press Next when you are finished.

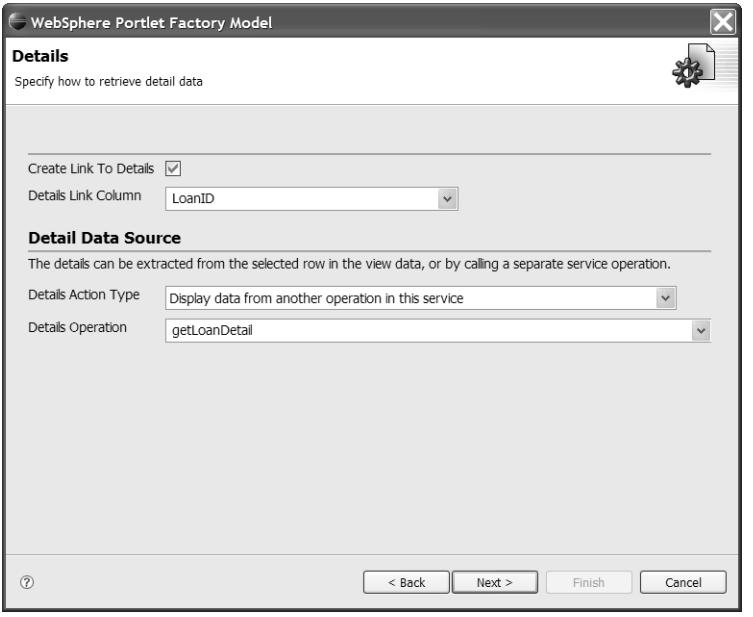


Figure 7.4 Configuring the detail portion of the loansList model.

Now, you need to define a uniquely identifying parameter that is used to open a loan when a loan ID is clicked. Enter in the loan ID from the loan on the currently selected row, as shown in Figure 7.5, and click the Next button.

Name the model `loansList` and make sure it is created in the directory models/chapter07, and then press Finish. This creates a new model called loansList in your WPF project, and opens the model in the Model Editor and Outline view. The loansList model contains three builder calls: a Portlet Adapter, a Service Consumer, and a View & Form.

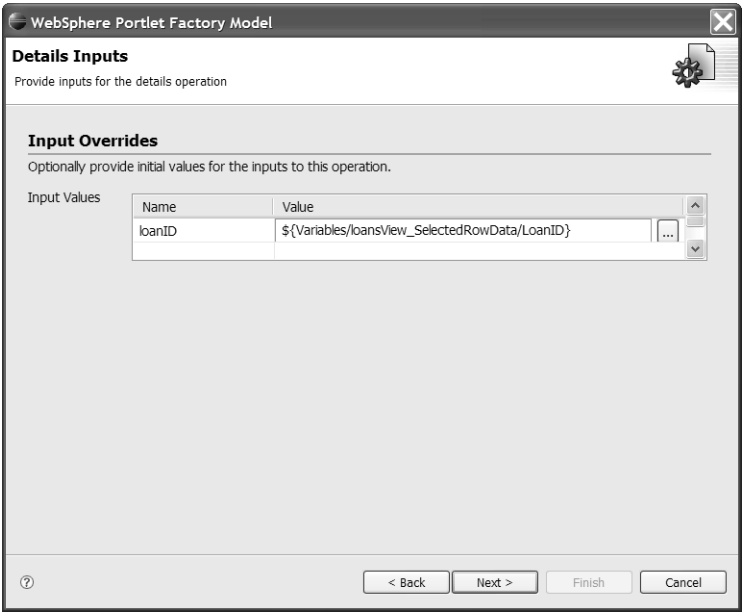


Figure 7.5 Configuring input overrides for the loansList model.

Configuring the Portlet Adapter

The Portlet Adapter builder converts the model into a portlet, which can be viewed inside a portal. Note that if you make any changes to this builder call, including the changes you are about to make, you need to rebuild the deployed application before you can see the results of the changes. For more information on this process, see the example in Chapter 1. The Service Consumer builder makes the operations in the loansService service available in the current model, and the View & Form builder displays the list and detail data in the model to the screen.

You need to make two small changes to the Portlet Adapter builder call. Double-click the Portlet Adapter builder call to open it in the Model Editor, and then change the Portlet Title to `Loans List` and the Portlet Description to `List of loans`. Save the builder call when you are finished. The Portlet Title identifies the portlet inside the portal—it is actually displayed in the title bar of the portlet—and the Portlet Description appears on various administration screens inside the portal.

Defining the Portlet as a Cooperative Source

The final step in building the list portlet is to add a Cooperative Portlet Source builder call. This builder call defines the structure and content of the information to send to the Property Broker (for instance, the loan ID). A Cooperative Portlet Target builder call is then used in the detail portlet to receive the loan ID from the Property Broker.

Add a Cooperative Portlet Source builder call to the loansList model and then name it `openLoanDetail`. Change the Type input to Property Broker Link, which replaces the links created by the View & Form builder with links that send the loan ID to the Property Broker. Note that the links are replaced only when the model is viewed as a portlet and the Property Broker is correctly configured; otherwise, the links created by the View & Form builder call are retained.

> **TIP**
>
> The Type input gives you four options for specifying a communication method. The first two, C2A Single Action and C2A Multiple Action, define click-to-action tags that can be placed on the page. Click-to-action is a feature of the WebSphere Portlet API, which has been deprecated in favor of the JSR-168 standard and is provided for backward compatibility only. The remaining two options define methods for publishing properties to the Property Broker. Whereas the Property Broker Link option automatically creates links on the page to communicate with the Property Broker, the Property Broker Action option creates an action that can be called from any control, such as a link, button, and so on. For more information, read the "Property Broker Action" section in this chapter.

In the Page location section of the Cooperative Portlet Source builder call, specify loansView_ViewPage as the Page and LoanID as the tag. This replaces the ID links created by the View & Form builder call with new links that publish information to the Property Broker.

For portlets to communicate via the Property Broker, they need to use the same namespace. In the example in this chapter, the Cooperative Portlet Source and Cooperative Portlet Target both need to have the same namespace defined. Leave the default value for the Namespace input.

The Output Definitions section defines the information that you send to the Property Broker. Fill out this section, as shown in Figure 7.6. Note that the Value input should read `${Variables/LoanLoopVar/Loan/LoanID}`. This declares that whenever a loan ID link is clicked, the loan ID of the selected loan should be sent as a String called loanID to the Property Broker. Save the builder call when you are finished.

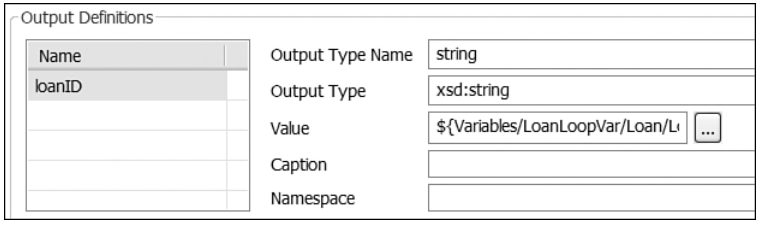


Figure 7.6 Configuring output definitions for the Cooperative Portlet Source builder call.

Your project now contains a list portlet. The next section describes how to test this portlet from your IDE.

Testing the List Portlet

Although you cannot test the communicative capabilities of the list portlet at this stage, because there is no detail portlet to communicate with, you should run the loansList model from the IDE to see if you are getting any errors. To do this, run the loansList model from the WPF Designer. You should see a list of loans in your default Web browser, as shown in Figure 7.7.

LoanID	Item	LoanedTo	DueDate
00001	The Gullible Traveler	Malcolm Johnson	11/05/2007
00002	A Traveling Aunt	Samuel Russell	11/11/2007
00003	An Interminable Journey	Joseph Jones	12/22/2007

Figure 7.7 List of loans from the loansList model.

Notice that the ID of each loan is a clickable link. If you click on the ID link for a particular loan, you should see some details for that loan. For example, clicking on the ID for A Traveling Aunt should return the details shown in Figure 7.8.

LoanID	00002
Item	A Traveling Aunt
LoanedTo	Samuel Russell
DueDate	11/11/2007

Figure 7.8 Testing the getLoanDetail operation.

Note that currently the list portlet functions as both a list *and* detail portlet, because the View & Form builder call provides links that make it possible to open details on a loan from the loans list. However, when you deploy the loansList model as a portlet and configure the Property Broker, these links are replaced with links from the Cooperative Portlet Source builder call. When the links from the Cooperative Portlet Source builder call are used, the detail page in the list portlet cannot open when a loan ID is clicked. Close the loansList model when you've finished testing it.

You now have a list portlet that provides a list of loans by consuming the loansService service. The next section describes how to create a detail portlet that displays details for a loan clicked in the list portlet.

Creating a Detail Portlet

This section describes how to build a detail portlet, which communicates with the list portlet created earlier. As with the previous portlet, this portlet consumes the loansService service to obtain its loan information. Details for loans selected in the list portlet are brought up in the detail portlet via the Property Broker—a screen shot of the portlet is shown later in Figure 7.15.

The model uses the following builders to provide its functionality:

- Portlet Adapter
- Service Consumer
- Page (x2)
- Action List
- Data Page
- Cooperative Portlet Target
- Event Handler

Creating a Model

The list model is a good starting point for building the detail portlet, so make a new copy of the list model by selecting the loansList model in the Project Explorer view, and then pressing Ctrl+C, then Ctl+V. Name the new model `loanDetail` when prompted.

Double-click the loanDetail model to open it for editing. Before you add builder calls to communicate with the loansList portlet, you should first modify the loanDetail model so it does not conflict with the loansList model when it is deployed as a portlet.

To do this, open the Portlet Adapter builder call in the loanDetail model, change the Name input to `loanDetail`, and then change the Portlet Title input to `Loan Detail`. This ensures that there are no conflicts between the list and detail portlets when you deploy them. Also, change the Portlet Description to `Loan details`. Now delete the Cooperative Portlet Source builder call from the loanDetail model, because in the current example the loanDetail model is a target of communication, not a source. Also, delete the View & Form builder because you are using a Data Page instead. You can reconfigure the View & Form so that it displays only detail information, but, in this case, it is easier to use a Data Page.

Tip

A single portlet can be both a target and a source of communication. For example, you might want a modification in the detail portlet (say, to the date of a particular loan) to be reflected in the list portlet. In this case, both portlets are sources and are both are targets; an ID is sent from the list portlet to the detail portlet, and then a refresh request is sent back to the list portlet after the detail portlet is updated.

Note also that a single portlet can define multiple sources and multiple targets. You might want to define multiple sources for a portlet if the portlet publishes more than one piece of information, and multiple targets are useful when you want to receive multiple properties from the Property Broker.

Adding a Default Message Page

Note that when the loanDetail portlet loads, it should not display any loan information, but rather a message indicating that no loan is currently selected. To bring this about, you need to add another Page builder call to the model. Add a new Page builder call, and name the builder call `defaultPage`. Enter in the following HTML into the Page Contents (HTML) input:

```
<html>
  <body>
      <div>
      No loan selected.
      </div>
  </body>
</html>
```

This displays the text `No loan selected`. in the Loan Detail portlet. Save the builder call when you are finished.

> **TIP**
>
> You should use only the Page builder for small snippets of HTML specific to your application. Any HTML that you might want to share between projects should be included in an HTML file and then imported into your model using the Imported Page builder.

Adding a main Action

Now, add an Action List builder call to the model. Call the action list main, which causes the action list to execute whenever the model is run. Enter `defaultPage` for the first action in the Actions input and save the builder call. The loanDetail model now displays the defaultPage whenever it runs.

Adding an Interface for Loan Details

Add another Page builder call to the model; this creates an HTML page that displays whenever loan details are to be displayed for a loan. Change the Name of the builder call to `detailsPage`, and then enter in the following HTML into the Page Contents (HTML) input:

```
<html>
  <body>
    <span name="loanDetails"></span>
  </body>
</html>
```

This page has a single span tag on it (loanDetails) that is overwritten with loan details using a Data Page. Save the builder call when you are finished, and add a Data Page builder call to the model to add the loan details. Type the name `detailsPage` for the Data Page, and then open the Select Action dialog for the Variable input. Select the Loan element from the results of the getLoanDetail operation, as shown in Figure 7.9, and press OK. This causes the Data Page to display the results of the getLoanDetail operation.

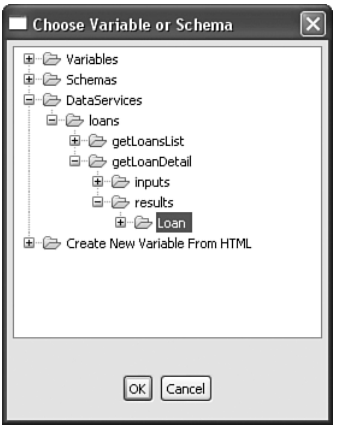


Figure 7.9 Selecting the Loan element.

Next, point the Page in Model input to detailsPage and change the Page Type to View Only so that there are no data entry controls inserted onto the page. Finally, specify detailsPage for the Location for New Tags input and save the builder call.

Defining the Portlet as a Cooperative Target

The next step is to define the loanDetail model as a target for inter-portlet communication. To do this, add a Cooperative Portlet Target builder call to the model. Change the Event Name input to `openLoanDetail`, and then fill out the Input Definition section, as shown in Figure 7.10. These settings define the communication inputs and should be the same as the outputs defined in the Cooperative Portlet Source builder call in the loansList model. Keeping these settings identical makes it easier to understand what is being sent where, and prevents possible type mismatches.

Leave the Output Definition section blank. There are no outputs for this builder call, because nothing is sent back after the loanDetail portlet receives the loan ID.

Notice the namespace at the bottom of the builder call. This builder must be the same as the namespace defined in the Cooperative Portlet Source builder call, or else the two portlets cannot communicate (you can leave the default value). Enter in `Displays loan detail` for the Caption and save the builder call. Note that this caption describes what happens when the property changes, whereas the caption in the Input Definition section describes the property itself.

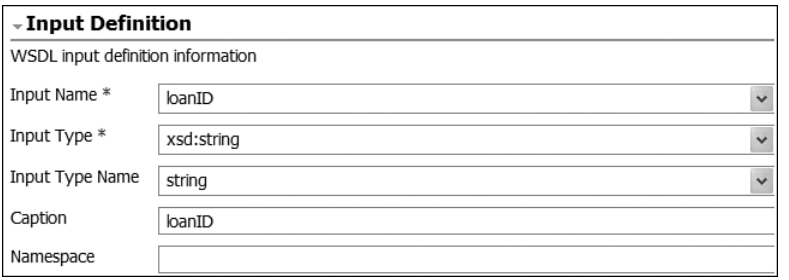


Figure 7.10 Configuring the Cooperative Portlet Target builder call.

Handling an Inter-Portlet Communication Event

You now have an event called openLoanDetail, which you later link to a loan ID being clicked in the loansList model. However, you haven't defined anything to happen when the openLoanDetail event is triggered. To do this, you need an event handler. Add an Event Handler builder call to the model and call it `handleOpenLoanDetail`. Select the openLoanDetail event from the Event Name drop-down box and notice that a String argument called loanID has been added in the Arguments section. This loanID is the same loanID as that specified in the Cooperative Portlet Target builder call, which means that you can now access the loan ID from the openLoanDetail event.

Change the Return Type input to void, because there is no need to return any information to the caller of this event. You simply want to perform some actions, which you specify in the Actions input. Open the Select Action dialog for the first action and find the getLoanDetail operation under the Methods section. Notice that there are two versions of the operation. Select the one that enables you to specify arguments, as shown in Figure 7.11, and press OK. A Define Method Call Arguments dialog appears; select the loan ID from under the Variables section in the Select Action dialog, which should then populate the value ${Arguments/loanID} to the underlying dialog. Press OK when you are finished to accept the new action.

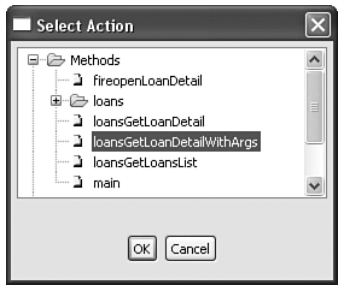


Figure 7.11 Adding the loansGetLoanDetailWithArgs method.

For the second action, select the details page from under the Page section of the Select Action dialog. It appears as 'detailsPage'. The actions are now complete: The first action consumes the getLoanDetail operation on the loansService service, passing in the loan ID as a parameter. This loads only the information into a results variable, so the second action is necessary to display the results. Save the builder call when you are finished.

Your project now contains a detail portlet. You will run a preliminary test on the detail portlet in the next section, and then test it in full after it has been deployed to a portal server and the Property Broker has been configured.

Testing the Detail Portlet

At this point, you should test that there are no obvious problems in the loanDetail model by previewing it from your IDE. You should see the message shown in Figure 7.12 displayed on the screen.

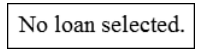


Figure 7.12 Testing the loanDetail model.

After you have configured the Property Broker later in this chapter, you can do a full test of the communication between the list and detail portlets. Before you do this, however, you should rebuild your application on the portal server. For instructions on how to do this, see the example in Chapter 1. Then, you can view the list and detail models as portlets. After you have added the portlets to a page in the portal, they should appear, as shown in Figure 7.13.

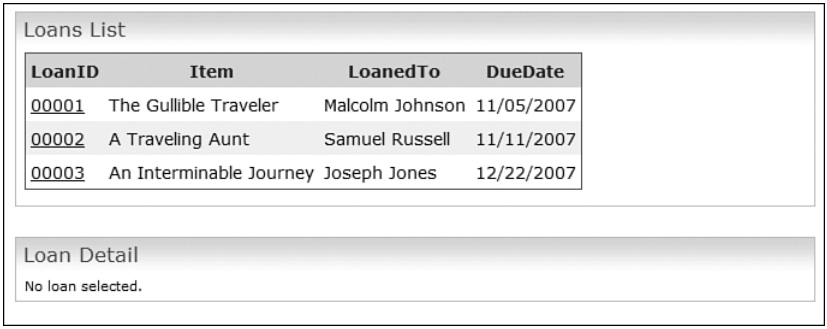


Figure 7.13 The loans application portlets.

Your application has now been successfully deployed to the portal server, although the portlets contained in the application cannot communicate with each other yet. The next section discusses how to set up this communication using the WebSphere Property Broker.

Configuring the WebSphere Property Broker

When setting up inter-portlet communication via the WebSphere Property Broker, some configuration is required to link or wire portlets together after they have been deployed. This section walks through the process of configuring the list portlet to send a loan ID to the detail portlet, which then displays details for the selected loan. At this point, note that the portlets are not wired together, so if you click on a loan in the list portlet, the details are opened within the same portlet, and the detail portlet still displays the `No loan selected.` message.

To configure the Property Broker, first log in to the portal as a user who has access to wire portlets together. You need a user with Privileged User access or higher to the portal page that is to contain the loan portlets. Navigate to where you added the Loan Detail and Loans List portlets and open the Page Menu for the page. You can do this in WebSphere Portal 6, for example, by moving the cursor to the right of the page title until a small arrow icon is revealed, as shown in Figure 7.14. Click the arrow to open the Page Menu, and then select Edit Page Layout.

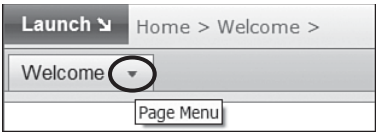


Figure 7.14 Opening the Page menu.

Click on the Wires tab, which should be the last tab on the Page Layout page. If you can't see the Wires tab, check that you are logged in as a user with at least privileged user access to the current page. This page lets you define inter-portlet communication for the portal. You can even define communication from one page to another. Note that you only define communication for portlets that have been added to a page in the portal. In this step, you add a row to the Wires page to define the communication between the list and detail portlet.

Select Loans List from the Source portlet dropdown, and then select loanID in the Sending dropdown. These values are taken from the Cooperative Portlet Source builder call, although they can be defined using IDEs other than WPF (WPF applications can communicate with non-WPF applications, and vice-versa). The current page is automatically selected as the target page. Select Loan Detail for the Target portlet, and set the Receiving dropdown to 'Displays loan detail, loanID'. This is the action caption and input name defined in the Cooperative Portlet Target builder call.

The final drop-down box gives you a choice of creating the wire as personal or public. A personal wire is accessible only by the current user, whereas a public wire is accessible to all users. Leave the default setting and press the ⊞ icon to add the wire to the portal. Wait for the page to reload and press the Done button to return to the portal.

You have now configured communication between the list and detail portlet.

Testing Inter-Portlet Communication

When you press the Done button on the Wires page, you are returned to where the list and detail portlets are displayed on the portal. To test that the communication is working correctly, click on a loan in the Loans List portlet. When the page reloads, a loan should be opened in the Loan Detail portlet, and the Loans List portlet should still display the loans list, rather than the loan detail, as it did previously. For example, clicking on 00003 should bring up loan details for The Interminable Journey, as shown in Figure 7.15.

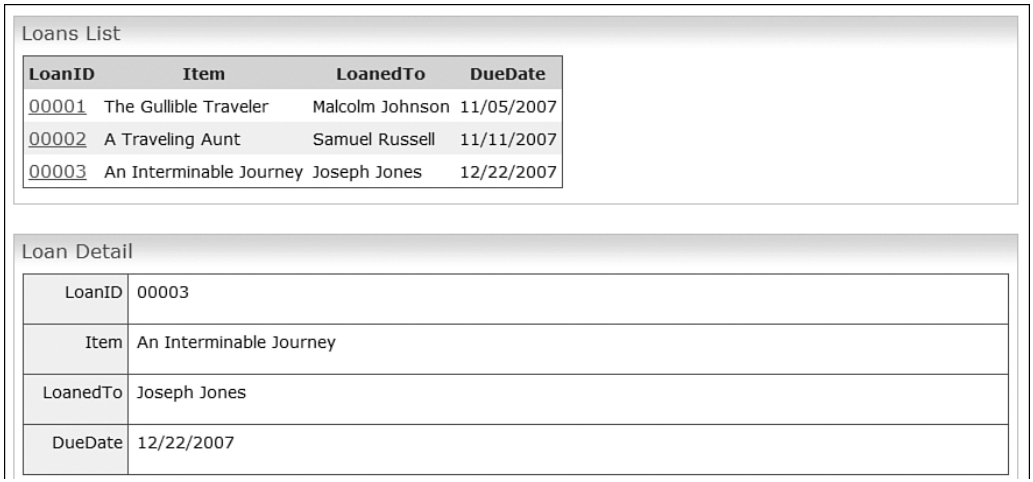


Figure 7.15 Testing inter-portlet communication in the portal.

You have now successfully created, deployed, and tested an application consisting of inter-communicating portlets. However, the method employed in this section is only one of several ways to set up inter-portlet communication. Some alternatives to this approach are discussed in the next section.

Alternative Communication Methods

Although the WebSphere Property Broker is a highly flexible and versatile method of inter-portlet communication, there are some situations where other approaches might provide more value. For example, the Property Broker is available only in WebSphere Portal, so if you transfer your portlets to another type of portal server, you need to modify the communication technique used. Second, communicating via the Property Broker requires that a wire must be set up between two portlets before they can communicate; this creates extra configuration steps and leaves more of the configuration process open to human error—although, it can also be regarded as a good thing, because it gives end users more control over the communication process.

There are several alternative options available for setting up inter-portlet communication in WPF, and each has its own strengths and weaknesses. These methods are described next.

Property Broker Action

Property Broker Actions, as opposed to Property Broker Links, let you define your communication as actions rather than as clickable links. The advantage of this approach is that you have more control over how the action is run—whether it be programmatically or from a UI control; however, the disadvantage is that more WPF configuration is required than when using Property Broker Links. The communication on the Property Broker is no different—that is, you still publish properties in the same way.

Property Broker Actions are not really an alternative method of communication to Property Broker Links, because they are really just Property Broker Links that don't produce a link in the UI. However, Property Broker Actions can be quite useful, and because the configuration process is sufficiently different, an example is included.

The following builder is added to the loansList model in this section:

- Link

Modifying the Cooperative Source

To alter the communication in the loans application to use Property Broker Actions, first open the Cooperative Portlet Source builder call in the loansList model. Change the Type input from Property Broker Link to Property Broker Action. This creates an action to publish the loan ID to the Property Broker, but it does not create the corresponding trigger for the action, such as a link. Save the builder call when you are finished.

Adding a Link and Configuring Communication

To trigger the Property Broker Action, add a Link builder call to the loansList model. Name the Link `loanIDLink`. In the Page Location section of the builder call, fill out the Page input as `loansView_ViewPage` and the Tag input as `LoanID`. This replaces the loan ID links created by the View & Form builder call with the new links defined by the Link builder.

Select pbAction_openLoanDetail for the Action input, which triggers the Property Broker Action when a loan ID is clicked. Notice that two arguments with the pbAction_openLoan Detail_Arg prefix are automatically added to the Input Mappings input in the Arguments section of the builder call. You need to replace the values of these arguments. The first argument to any Property Broker Action must be the name of the action (that is, pbAction_openLoanDetail). This argument is used by the portal to associate the Property Broker call with a particular Property Broker action. The argument is added automatically when using Property Broker Links, but needs to be added manually when using Property Broker Actions. Rename the argument to `ACTION_NAME`, and then type the name of the action (`pbAction_openLoanDetail`) as the value. Note that the name and value of the argument must be written as specified, or the inter-portlet communication cannot work. Note that the Evaluate Arguments input must be set to 'As the page is rendered'; if it is not, the links cannot be generated correctly.

Because the Cooperative Portlet Source builder call is expecting an argument, you must also change the second argument for each link created by the Link builder call. To do this, change

the second argument to point to the currently selected loan, and change the name of the argument to `loanID` so that it is easier to identify, as shown in Figure 7.16. Note that the value of the argument is the same as the value of the argument specified in the Cooperative Portlet Source builder call. In the next step, you remove the value from the Cooperative Portlet Source builder call so that the value from the Link builder call is used instead. Save the builder call when you are finished.

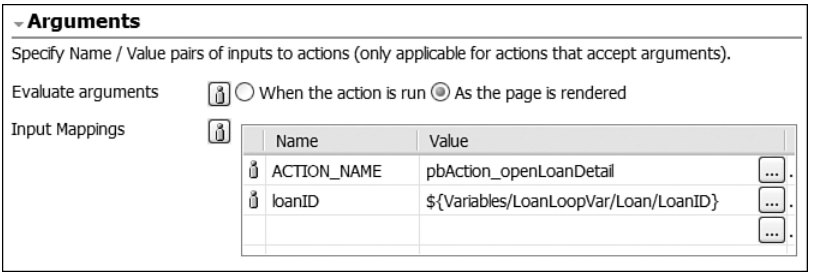


Figure 7.16 Adding an argument to the Link builder call.

Open the Cooperative Portlet Source builder call and delete the Value input for the loanID output. The value for the loanID output is now taken from the Link builder call.

You have now finished configuring the loansList model. No changes are required to the loanDetail model, because the parameter published to the Property Broker is the same as it was before. Note that the ACTION_NAME parameter does not need to be specified in the Cooperative Portlet Source builder call. Rebuild your portlet application by right-clicking on your project and selecting Portal Server WAR, Build Portlet War. You don't need to reset or modify the wire, because the structure of the communication hasn't changed. The only thing that has changed is how you actually trigger the event to send to the Property Broker. To test the new communication method, navigate to the loans portlets in the portal and click on a loan in the Loans List portlet. The corresponding loan should open in the Loan Detail portlet.

You have now successfully configured the loans application to use Property Broker Actions rather than Property Broker Links. Again, note that this change doesn't change the type of communication, but merely how you use it in WPF.

WPF Event Model

When using the WPF event model, communication between portlets occurs by triggering and handling an event. This approach is easy to use and configure: First, you declare an event in a source and target portlet, and then you trigger, or fire, the event in the source portlet. Finally, you handle the event in the target portlet. Events can be fired and targeted to a particular portlet,

or they can be fired in such a way that they can be accessed by any other portlet in the same WAR file.

You can trigger events using the WPF event model in two ways: on the server and on the client. Both methods enable you to access and manipulate resources on the server, such as reading and updating data from a service provider, but triggering events on the client also enables you to perform partial page refreshes. Partial page refreshes improve the speed of your application, and users appreciate that they can interact with your portlet without having to constantly reload the entire portal page. Partial page refreshes are covered in more detail in Chapter 13, "Using Ajax and Dojo." Note that you can only pass parameters to events when using server-side events, which is why both types of events are sometimes used together; for example, you might use a server-side event to update data in a Lotus Notes database, and then use a client-side event to partially refresh the page with this data. The example at the end of this section uses only a server-side event.

Be aware that when using the WPF event model approach, all inter-communicating portlets need to be contained in the same WAR file, which also means that they all need to be built with WPF. As a result, the WPF event model is most useful for setting up inter-portlet communication when interoperability with other applications is not a concern.

The following builders are added in this section:

- Event Declaration (x2)

Defining an Event

To alter the communication in the loans application to use the WPF event model, the first step is to define an event, which is triggered every time a user clicks on a loan ID. Add an Event Declaration builder call and change the Event Name input to `openLoanDetail`. Add an argument called `loanId` of type String to the Arguments section so that any triggers for the event need to pass in a String value when they trigger the event. This String corresponds to the loan ID of the loan that the user clicks on. The event is triggered any time details for a loan are opened. Save the builder call when you are finished.

Triggering the Event

Now that you have finished configuring the event declaration, you should modify the loansList model to trigger the event. Open the Link builder call in the loansList model and change the Action input to fireopenLoanDetail. You can select this from the Select Action dialog. If the fireopenLoanDetail action does not appear, make sure you have saved your Event Declaration. The fireopenLoanDetail action fires the openLoanDetail event instead of publishing the loan ID to the Property Broker.

Tip

If you want to fire an event to a specific target, you can use the fireTargeted trigger. For example, to fire an event to the loanDetail model so that it can't be accessed by other models, select fireTargetedopenLoanDetail instead of fireopenLoanDetail. You are prompted for the target of the event, and you can specify four possible targets:

${Java/WebAppAccess.EVENT_TARGET_ALL})

This fires the event to all models in the current application.

${Java/WebAppAccess.EVENT_TARGET_PARENT})

This fires the event to the parent model of the current model, which is any model that is used to load the current model (for example, if you're using a Linked Model builder call).

${Java/WebAppAccess.EVENT_TARGET_SELF})

This targets the event to the current model.

${Java/WebAppAccess.getModelInstance(targetModel)})

This targets the event to the targetModel model. You should substitute the text targetModel with the actual name of the target model—for example, loanDetail.

Notice that an argument called fireopenLoanDetail_Arg1 has been added underneath the Arguments section of the builder call. This argument has been created by WPF to cater for the loan ID that is supposed to be passed to the event, but you still have two arguments left over from when you configured the Property Broker Action. The loanID argument left over from the Property Broker Action is the argument you want to pass to the event when it is fired, so remove the other two arguments from the Input Mappings input. The Arguments section should now appear, as shown in Figure 7.17.

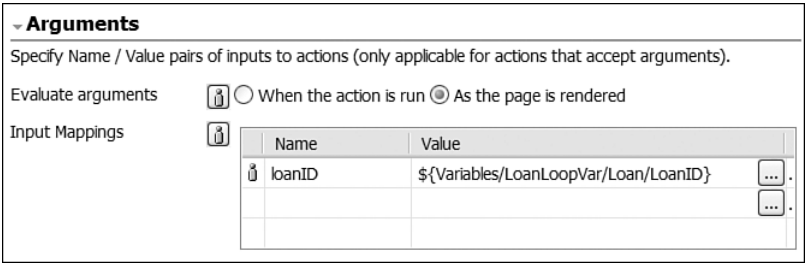


Figure 7.17 Setting arguments for the Link builder call.

You have now finished configuring the Link builder call. Disable the Cooperative Portlet Source builder call as it is not being used, but you might want to use it again later. You can disable builder calls by right-clicking on them and selecting Disable. Save the model when you are finished.

Handling the Event

Now, you need to configure the loanDetail model to handle the openLoanDetail event. To do this, first open the loanDetail model and disable the Cooperative Portlet Target builder call, because you are no longer using the Property Broker (don't save the model yet). Copy the Event Declaration builder call from the loansList and paste it into the loanDetail model. This is done so that both models are using the same event. Because the event name in your Event Declaration is the same as the event name in your Cooperative Portlet Target builder call, no changes are required to the Event Handler builder call; it processes the openLoanDetail event as it is defined by the Event Declaration now that the Cooperative Portlet Target is disabled.

Save the model to complete the configuration of your application to use inter-portlet communication via events. To test the new communication method, rebuild your application and navigate to the loans portlets in the portal. Click on a loan in the Loans List portlet, and the corresponding loan should open in the Loan Detail portlet.

Shared Variables

Shared variables are perhaps the quickest and easiest way to implement inter-portlet communication. Shared variables can be stored in one of four ways: in the HTTP session, in the HTTP request, in the JVM where the current WebApp is running, or using a custom Java class. Variables stored in the JVM are accessed as static variables (the same copy is used across all instances of the portlet). To create or read a shared variable from a model, add a Shared Variable builder and designate a variable to share.

Note that variables stored in the HTTP session increase the amount of memory consumed by the session, and variables stored in the HTTP request have limited scope. Also, variables in the JVM cannot be accessed outside the JVM (for instance, across a cluster), and custom variable storage methods require additional Java development. Provided you keep these concerns in mind, however, shared variables are a powerful and easy-to-configure method of inter-portlet communication.

In the previous section, it was necessary to pass the loanID from the event trigger to the event handler via an argument to the event. In this section, you use a shared variable in place of the loanID argument so that when you change the loanID variable in the loansList portlet, it is read in the loanDetail portlet and used to update the loan information. Note that in this example, an event is still used to access loan details for the loan ID stored in the shared variable. This is because even though the loan ID is shared between both models, the details portlet won't know that it is supposed to refresh the other loan details unless an event is fired that causes this to happen. Because changes to a shared variable do not in themselves trigger any events, shared variables are often used in conjunction with the WPF event model.

The following builders are added in this section:

- Variable (x2)
- Shared Variable (x2)
- Action List

Removing the Event Argument

To alter the communication in the loans application to use the shared variable approach, first remove the argument from the Event Declaration builder call in both the loansList and loanDetail models, because you are now storing the loan ID in a shared variable. The argument is listed in the Arguments input of the Event Declaration. Save both instances of the builder call when you have removed the arguments.

Adding a loanID Variable

Next, add a Variable builder call to the loansList model and call it `loanID`. Change the Type input to String and save the builder call. This builder call holds the value of the currently selected loan ID and is used in both models, so copy and paste the builder call into the loanDetail model. Save the model when you are finished.

By default, each model uses its own instance of the variable created by the Variable builder call. To share the variable, you need to add a Shared Variable builder call to each model. Add a Shared Variable builder call to the loansList model first, and then name the builder call `loanID-Shared`. Change the Variable input to `loanID`.

The Scope input enables you to specify where the shared variable will be stored. As discussed earlier, you have four options when setting a shared variables scope: Session, Request, Application, and Custom. Make sure this input is set to Session, which means that the variable value persists even when other links on the page are clicked. Then, change the Unique ID input to `loanID` and save the builder call. Make a copy of the Shared Variable builder call and paste it into the loanDetail model. Save the model when you are finished.

Creating these Shared Builders shares the loanID variable in the HTTP session using the text loanID as an identifier. The identifier can then be used from other processes and portlets to access the shared variable from the HTTP session. When referring to the variable from WPF, however, you need only to reference the name in the Variable builder call.

> **TIP**
>
> When referring to a shared variable in other builders, refer to the Variable builder call rather than the Shared Variable builder call.

Configuring Actions for loansList

The next step is to set up two actions in the loansList model. The first action changes the value of the loanID variable when a loan ID is clicked, and the second action fires the new openLoanDetail event without any arguments.

To add these actions, add an Action List builder call to the loansList model and call it `selectLoan`. Enter in an argument called `loanID` of type String in the Arguments input, and then change the Return Type input to void to prevent the Action List from returning a value.

For the first action, select Assignment from under the Special heading in the Select Action dialog. For the Target variable, select the loan ID variable (Variables/loanID), and for the Source variable, select the loanID argument from the Action List (${Arguments/loanID}). The Make Assignment dialog should appear, as shown in Figure 7.18. Press OK when you are finished to accept the action.

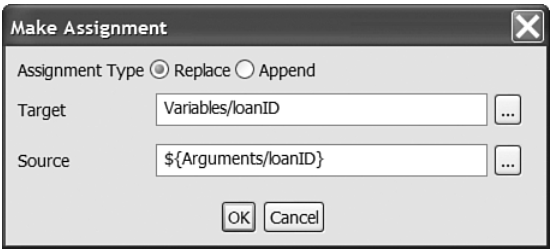


Figure 7.18 Configuring the Make Assignment dialog.

For the second action, fire the openLoanDetail event by selecting fireopenLoanDetail from the Select Action dialog. Save the builder call when you've finished adding both actions.

Running selectLoan

Open the Link builder call and change the Action input to `selectLoan`. This runs the selectLoan action list whenever a loan ID is clicked. WPF assumes that you want to set a new argument for the selectLoan action list, so it automatically adds an extra argument to the Input Mappings input. You can delete the extra argument, because the previous loanID argument will suffice. Save the model when you are finished.

Using the Shared Variable in the loanDetail Model

You need to configure the loanDetail model to use the shared variable instead of the argument in the old openLoanDetail event. To do this, open the handleOpenLoanDetail Event Handler builder call and change the first action in the Actions input to use the loanID variable instead of the loanID argument, which no longer exists. The new action should read loansGetLoanDetailWithArgs(${Variables/loanID}).

Save the builder call when you are finished. To test the new communication method, rebuild the application and navigate to the loans portlets in the portal, and then click on a loan in the Loans List portlet. The corresponding loan should open in the Loan Detail portlet.

You have now successfully configured the loans application to use shared variables for its inter-portlet communication.

Click-to-Action (C2A)

Click-to-action (C2A) facilitates inter-portlet communication through the use of portlet menus that the user can configure. The C2A builders have been deprecated as of WPF 6.0, because they

rely on the now deprecated WebSphere Portlet API, rather than the JSR-168 standard, so you should avoid using C2A as your inter-portlet communication mechanism.

When to Use Inter-Portlet Communication

When developing portlets, in WPF or otherwise, spreading an application's functionality across several inter-communicating portlets can help to provide a more customizable interface and ultimately produce more extensible applications. However, some scenarios are more conducive to inter-portlet communication than others. For example, you might not want to separate the list and detail portions of a data access function if they are going to be used on a page that already contains a list and detail portlet, because multiple list and detail portlets on the same page can clutter and complicate the interface. Similarly, if a list portlet depends on another portlet on the page for its information, you might want to include the list as a second page in that portlet, rather than as a separate portlet. The decision as to whether to combine functions into a single portlet is also influenced by how many other portlets you expect will be used on the same page and how much space they will take up. If you expect screen real estate to be scarce, perhaps you need to cater to 800x600 resolutions, and if you expect multiple portlets to be used at once, it might be best to economize and combine several functions into a single portlet wherever possible, rather than implement the same functions using inter-portlet communication.

Having said this, inter-portlet communication is a powerful tool for portlet developers to improve the usability of their applications and should be used wherever possible. As a general rule, potentially reusable, high-level functions, such as the list and detail components of a data portlet, are the best candidates for inter-communicating portlets, because they can be easily incorporated into other applications. The information sent between each portlet is fairly simple—usually just a uniquely identifying key for a particular record or document—which reduces the information sent between the server and the client, and therefore also speeds up your application. It is also the sort of information that could be meaningful to other functions or processes. Clicking on an item in a list portlet, for example, could trigger other portlets on the page to open charts, reports, or edit forms for that item.

Summary

In this chapter, you learned about the different approaches to implementing inter-portlet communication in WPF, in addition to the strengths and weaknesses of each approach. You also created a library loan system that consisted of a service provider and two portlets, which demonstrated each approach. When you clicked on a loan in the list portlet, it caused details on that loan to be opened in the detail portlet. Both portlets retrieved their data from the service provider.

The next chapter, "Using Java in Portlets," discusses how to utilize Java code in your WPF applications.

Important Points

- WPF portlets can communicate with each other using the WebSphere Property Broker, shared variables, or the WPF event model. Each approach has its own strengths and weaknesses.
- Communication via the WebSphere Property Broker is facilitated by publishing properties to a WebSphere Portal Server mechanism known as the Property Broker, which then routes these properties off to other portlets. Implementing communication using the Property Broker offers considerable flexibility, because it means that your portlets can communicate with portlets in other WARs, or even portlets built using environments other than WPF. Also, because WPF provides builders to automate the communication configuration, you don't need to write any code to implement the communication. However, configuration in the portal is required before this mode of communication will work.
- Shared variables can be configured to use a number of different stores and are perhaps the quickest and easiest way to implement inter-portlet communication in WPF. However, unless your portlets are all in the same WAR file, you need to write code to retrieve the variable values. This approach is best suited to storing one or two small pieces of information used across an entire WPF application.
- The WPF event model also offers a quick and easy approach to inter-portlet communication, and gives you the added benefit of a prepackaged WPF builder to handle communication events without regenerating the entire portal page. This approach has limited extensibility because all communicating portlets need to be contained in the same WAR file to be used effectively, and they need to be developed in WPF; however, it is well suited to WPF applications where events in one portlet trigger events in another.

Index

B

C

D

E

F

G

H

I

J

K–L

M

Q–R

S

T

U

V

W

X–Z