Getting started with WebSphere Portlet Factory V6.1

WebSphere Portlet Factory Development Team

29 July 2008

Abstract

Discover what the IBM® WebSphere® Portlet Factory can do for your portal development team. Learn key Portlet Factory concepts, and how to perform primary tasks, create data services, develop portlet user interfaces, techniques for debugging, and best practices for deployment. Put it all together to expedite and automate much of the production of the components in your portal interface.

The 6.1 release of this guide includes information on the new features of the 6.1 and 6.0.2 releases, such as REST service support and the new graphical Design view.

Table of contents

IBM WebSphere Portlet Factory overview

This section provides a high-level overview of the key features of WebSphere Portlet Factory and the underlying technology. It then walks through an example of how to use this product to create a portlet.

Product overview

WebSphere Portlet Factory is a powerful and flexible tool for rapidly building portlets on top of a service-oriented architecture. Developers are able to quickly and easily leverage their company's core assets, automatically assembling them into custom, high-value portlets. Portlets created with WebSphere Portlet Factory are dynamic, robust Java 2 Enterprise Edition (J2EE) applications that react automatically to change, and can be further modified by business users in real time, to meet changing business requirements without requiring any coding, duplicating or versioning of assets. By eliminating the need to code all of these implementations and their variations, WebSphere Portlet Factory simplifies the development, deployment, and change management process, saving companies time and money.

The primary features of WebSphere Portlet Factory are as follows:

- **Multi-page custom portlets without coding**. The WebSphere Portlet Factory rapid application development (RAD) capabilities and ease of use enable developers of all skill sets to create multi-page, complex portlets. Developers build portlets by defining a sequence of highly adaptive software components called builders, which have an easy-to-use interface. Developers assemble builders into models, which then generate the application code. In this way, developers can capture and automate the process of building dynamic portlets, instead of explicitly coding each portlet.
- **Robust integration capabilities with enterprise applications and data**. WebSphere Portlet Factory provides automatic integration with existing applications and data including SAP, Lotus Domino, PeopleSoft, Siebel, Web Services, relational databases, and Excel. Developers can quickly create composite, high-value portlets that leverage existing investment in your existing applications and data.
- **Automatic integration with WebSphere Portal**. With WebSphere Portlet Factory, you have direct integration with IBM WebSphere Portal features such as portlet wiring, Click-to-Action, business user configuration, people awareness, WebSphere Portal groups, and the credential vault. Portlets are deployed automatically to WebSphere Portal software.
- **Support for SOA development**. WebSphere Portlet Factory provides powerful technology for speeding the creation of service-oriented applications and portlets. It includes data services builders along with data integration builders that together automate the process of creating services from systems such as SAP and Lotus Domino. This services approach provides a clean way to separate the back end services of an application from the presentation layer. It also automatically creates testing support for back end services, and it enables front end presentation development and testing without requiring a back end connection.
- **Many portlet variations from a single code base**. With the profiling capability of WebSphere Portlet Factory, developers can easily create multiple portlet

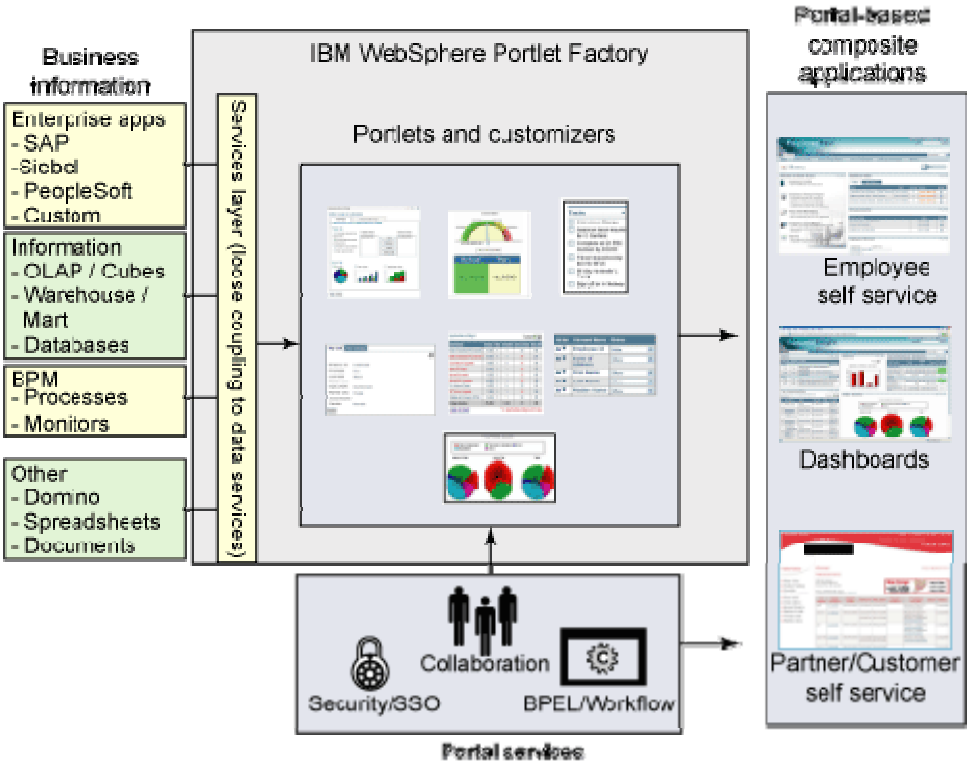
variations from one code base, without requiring any additional code changes or redeployment.

- **Automation of frequently occurring development tasks**. By creating new builders, developers and architects can capture commonly used design patterns and company-specific business processes as reusable components for all developers, enforcing application architecture and development best practices.

Technology overview

This section provides a technology overview of WebSphere Portlet Factory, starting with the architecture, which is illustrated in Figure 1.

Figure 1. WebSphere Portlet Factory application architecture



The first layer of the architecture stack pictured above is the data sources on the left. Data can be sourced from a number of different systems, including databases, enterprise applications such as SAP, collaborative platforms like Domino, and historical or analytical data from products such as SAP Business Warehouse.

The next two layers – services and portlets – can both be developed with WebSphere Portlet Factory Designer. The Designer tool is an Eclipse plug-in and runs seamlessly in Eclipse-based products, such as Rational Application Developer.

The last layer of the diagram illustrates the Portal server framework, which provides key services such as page navigation and creation tools, single-sign-on capabilities, user management, built-in process server, and collaboration features such as instant messaging.

Key concepts: builders, models, and profiles

With WebSphere Portlet Factory, developers build portlets by snapping together a sequence of components called *builders*. Each builder has a simple wizard-like user interface and does the work of automatically generating or modifying part of an application. A builder implements an application design pattern.

Builders are assembled in *models,* which are like application production lines. Each time a change is made to any of the builders in a model, the application code is regenerated, allowing the developer to iteratively develop a custom portlet application. The builders generate all the application code, including JSPs, Java classes, and XML documents.

In addition, developers can create multiple variations of a portlet from one code base, without requiring additional code changes or redeployment. This is done with the *profiling* feature of WebSphere Portlet Factory. Different profiles can be created for different user constituencies, based on any characteristics such as region, role, or group membership. The profiling technology is also used to support runtime configuration, so that business users can control application functionality through a simple browser interface. The net result is that WebSphere Portlet Factory allows companies to rapidly create adaptive applications that respond to change on demand – something traditional tools and technologies simply cannot provide.

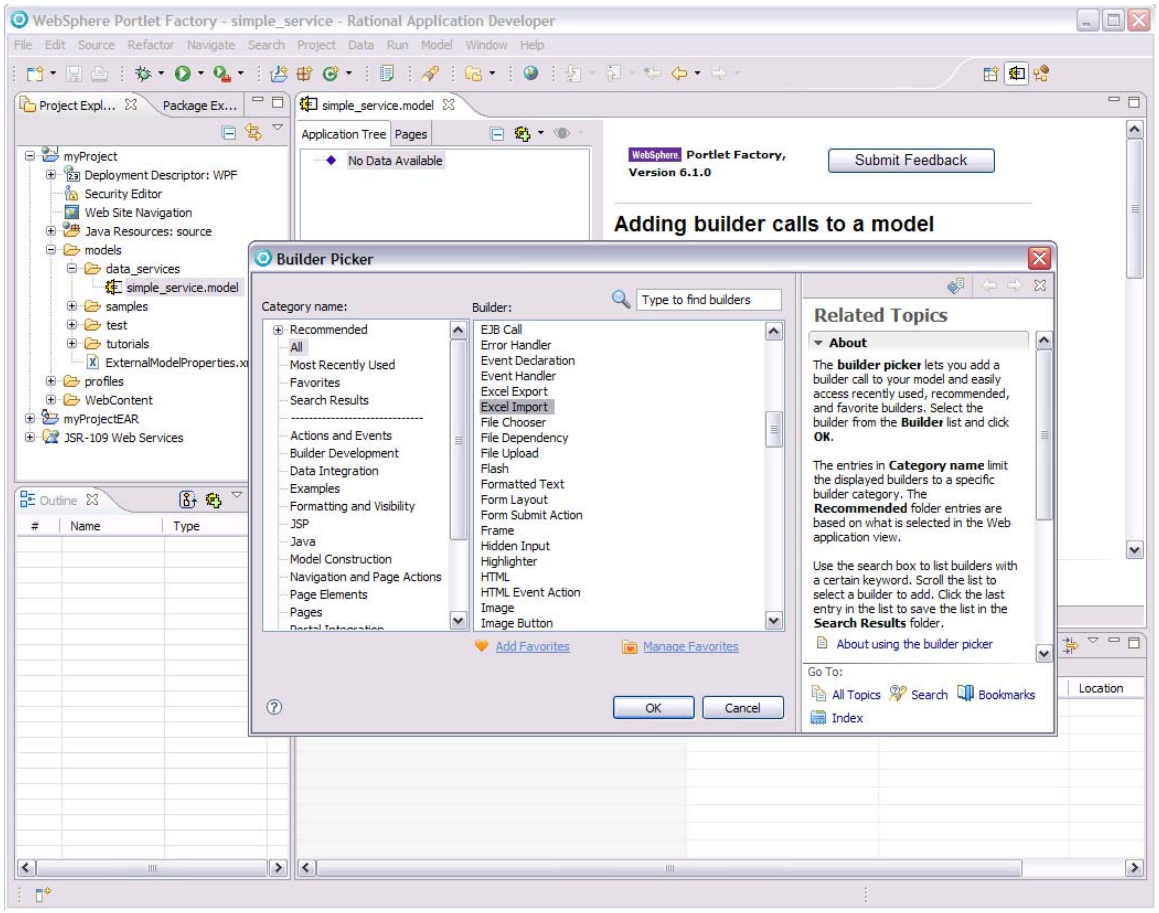
Using builders to create portlets or services

Developers use WebSphere Portlet Factory models to implement a portlet user interface or a data service. After creating a new model, the next step is to start adding builders to the model, which will create the desired application code (see Figure 2).

As described briefly above, builders are software automation components that capture design intelligence and automate the creation of code. Similar to customizable robots in an assembly line, builders perform specific software automation tasks, based upon inputs or parameters specified by developers. WebSphere Portlet Factory includes more than 130 builders that automate a wide range of tasks, such as creating HTML from a schema or integrating with common back-end systems (like Domino, SAP, Siebel, databases, and so on).

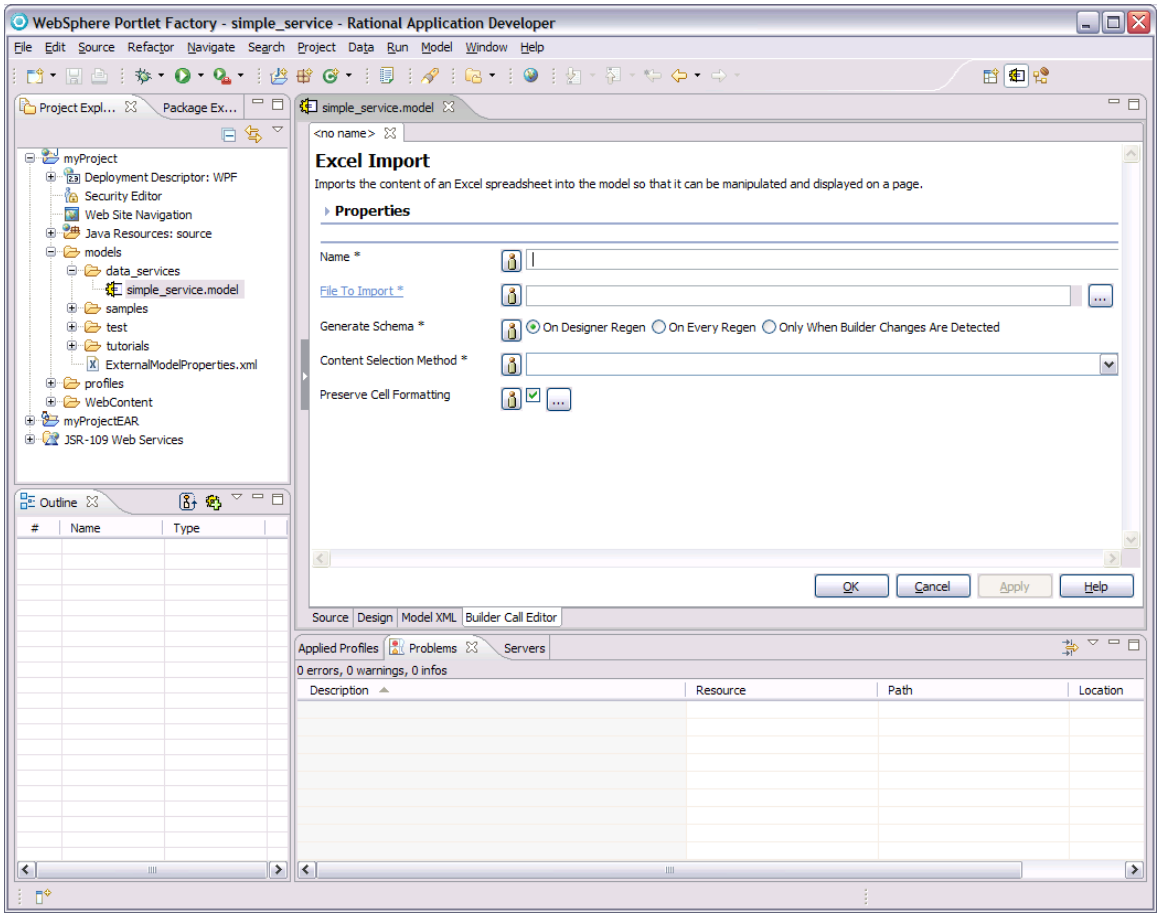
Builders are selected from the Builder Palette, which lists all the builders by category. The Builder Palette is shown in Figure 2.

Figure 2. WebSphere Portlet Factory Designer – adding builders to a model



When a builder is selected from the palette, you are presented with the builder call editor (see Figure 3). The next step is to provide the builder's inputs by typing in the appropriate inputs and using the builder's various pickers and selection widgets.

Figure 3. Editor for the Excel Import builder

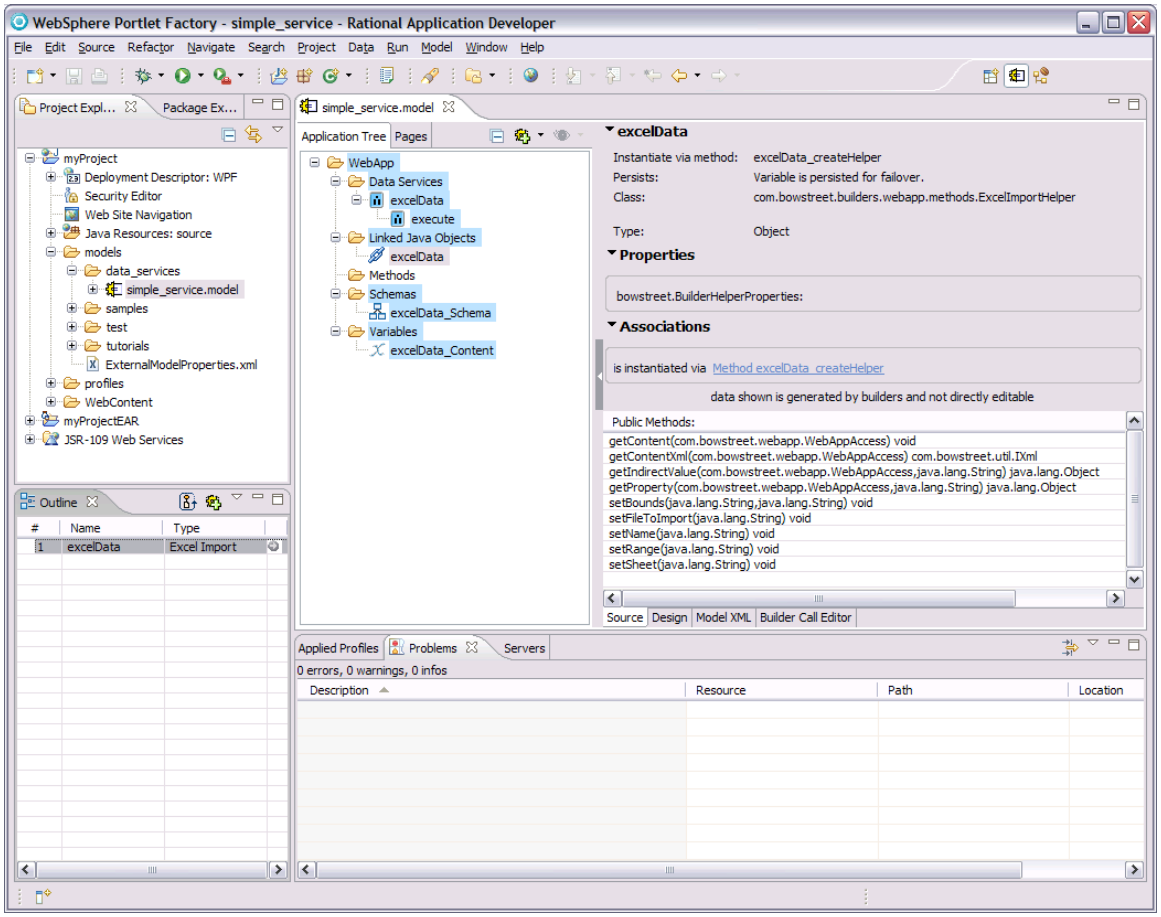


After the builder inputs are populated, when the OK or Apply button is clicked, the builder generates the appropriate code through a process called regeneration. When a model is regenerated, each builder in the model executes in sequence and creates pieces of your application, such as JSP pages or Java methods. Each time you change a builder input and click OK or Apply, regeneration occurs. Regeneration is similar to the process of “recalculating” a spreadsheet, the difference being that WebSphere Portlet Factory regenerates the portlet code, rather than a set of numbers.

You can see all of the code generated by the builders in your model by using the Application Tree view. To see the code generated by a particular builder, click on the arrow in the right-most column of the Outline view (the “Link to WebApp Tree” button). Figure 4, shows the model after the Excel Import builder has been added, and the Link to WebApp Tree button has been pressed. As a result, the Application Tree view highlights the code created by that builder, which includes a schema that describes the Excel data, an XML variable that will hold the data, and a data service that can be invoked to retrieve the data and place it in the variable.

Figure 4 shows the Application Tree view with the code created by the builder in the model.

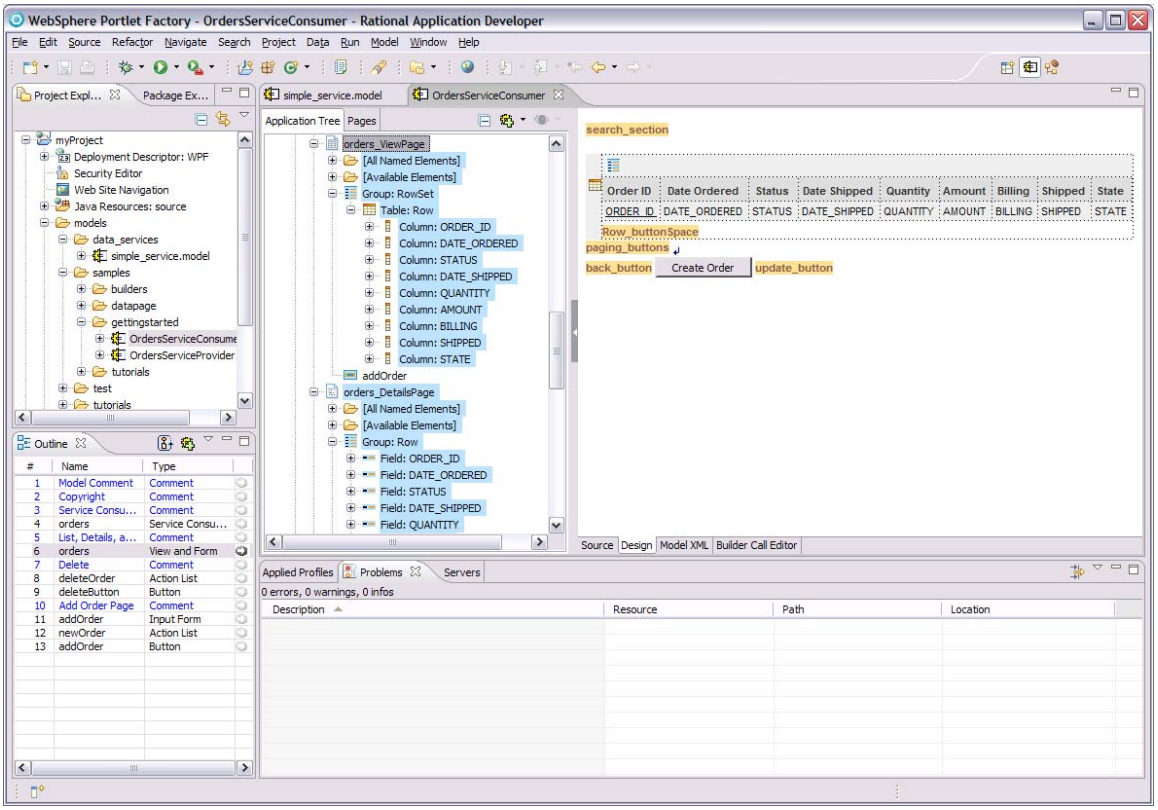
Figure 4. Application *Tree view*



Development continues by adding builders to your model until you have the desired functionality. Along the way, you can test the model by running it from the Designer tool, using any application server to see the intermediate results. You do not need to deploy to the portal unless you need to test portal-specific features such as portlet-to-portlet communication.

When working with pages and user interface elements of an application, the graphical Design view provides an up-to-date WYSIWYG view of your pages. You can select pages by clicking in the Application Tree or in the Pages tab. From the Design view, you can right-click on elements to bring up a list of suggested builders that can be applied to the selected page element.

Figure 5. Design view for one page of a portlet model



Development with WebSphere Portlet Factory is an iterative process. If you do not like what a builder has created, you can go back, change the builder's input values, and have the entire portlet application update instantly. Or, you can add additional builders to your model that modify what other builders have created. You can also import your own Java code, HTML, style sheets, and JSPs to get the exact functionality desired. In short, WebSphere Portlet Factory supports rapid prototyping, and it also includes all the necessary capabilities for you to turn your quick prototype into a production-ready portlet.

Dynamic profiling

Often, companies want to be able to tailor their portlets for different users, based upon their roles, personal preferences, or other types of contextual information. To implement customized portlets or portlets that adapt based upon information about the user, developers typically have had to write conditional logic or copy and paste the portlet code, creating versions for each unique variation required. These approaches to creating customized portlets are usually cost-prohibitive and, ultimately, a maintenance nightmare.

A key feature of IBM WebSphere Portlet Factory is *dynamic profiling*. Profiling solves the maintenance and scalability challenges that companies face when they try to build portlets that are customized for personal preferences, roles, groups, geography, and brand for individual users. With profiling, developers can very easily build portlets that automatically adapt and tailor themselves to the preferences and characteristics of each person using the portal.

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

Deployment overview

When your model is complete, you deploy it to the portal. To do so, simply add one additional builder to your model called Portlet Adapter. This builder will indicate that you want to deploy your model as a portlet. (Even with the portlet adapter in your model, you can still run it standalone and as a portlet.)

Creating a WebSphere Portlet Factory project

The first step for developing with WebSphere Portlet Factory is to install the product and configure a new project.

Step 1: Install WebSphere Portlet Factory.

Install WebSphere Portlet Factory. Run the installation program as documented in the installation guide provided with your product. The guide appropriate to your product version is located at the top level of the IBM CD. WebSphere Portlet Factory supports application deployment on WebSphere Portal Server.

Step 2: Create a project that includes WebSphere Portlet Factory features.

After installing WebSphere Portlet Factory, the next step is to create a Web application project. A project is the foundation of a portlet or application, and contains all the artifacts required to build the application. The easiest way to get started is to follow the ***Quick Start with WAS CE*** tutorial included with WebSphere Portlet Factory.

1. Select **Start → Programs → IBM WebSphere → Portlet Factory → Tutorials**.
2. Select **Quick Start with WAS CE**.
3. Use the instructions to start WebSphere Portlet Factory Designer and to create a new project (up to Step 3 in the tutorial).
4. When you reach Step 4 in the tutorial (adding the Tutorial and Samples Feature set), also add feature sets for the specific data sources you want to use. Note that support for relational databases does not require any additional feature set. Here are some other feature sets for data integration:
    - Lotus Collaboration Extension (for Lotus Domino access)
    - PeopleSoft Extension
    - SAP Extension
    - Siebel Extension
    - Excel Extension
    - Lotus Web Content Management Extension

    Tip: When you create your own applications in the future, add only those feature sets that you will actually use in your project. This reduces the size of the portlet WAR file you are creating. If you want the samples feature set, consider creating a separate project in Eclipse for these.

5. Complete the rest of the tutorial and follow the instructions there to verify successful deployment. Make sure that you can run some of the sample models and that you can make a change to a model and see your change when you run the model again.

Creating data services

After you have installed WebSphere Portlet Factory and created your project, the next step is to build the data services needed by your portlets. With the rich set of services builders included in WebSphere Portlet Factory, it is very easy to create portlets based upon a Service-Oriented Architecture (SOA).

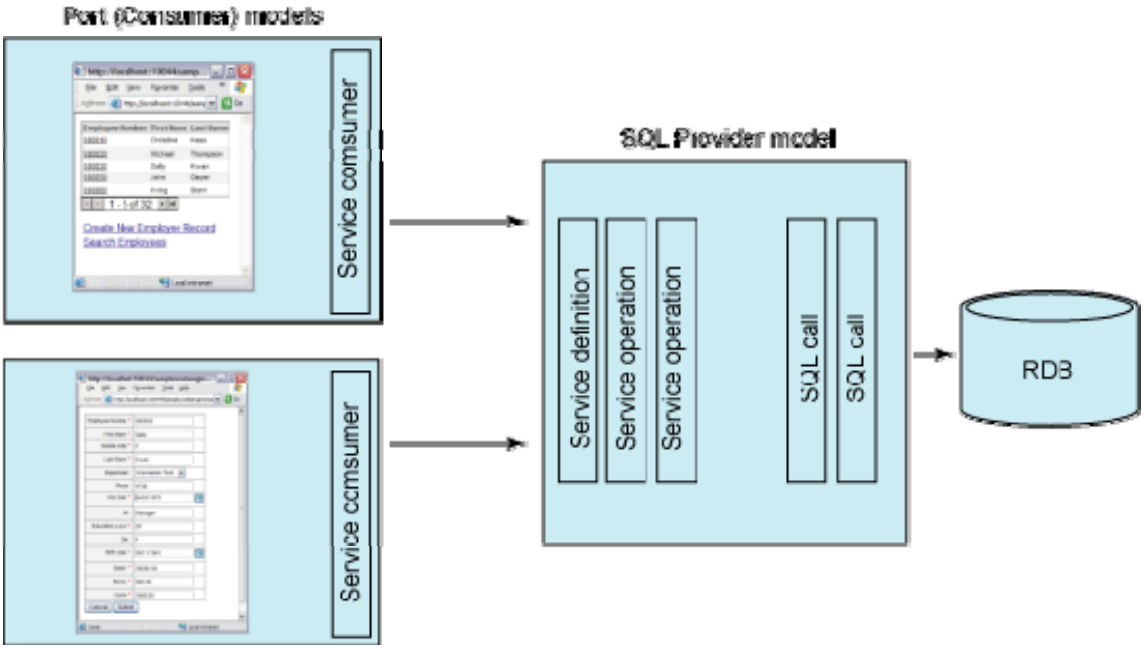
Why use WebSphere Portlet Factory builders for SOA

With WebSphere Portlet Factory, you can quickly build portlets where the data and the presentation layer are all in one model. For example, you can very quickly create an SAP portlet by creating a new model and then adding the SAP View & Form builder to your model. This one builder allows you to quickly connect to an SAP BAPI and then generate input, results, and even detail pages. While this approach is very fast and efficient, it is not the optimal architectural approach.

A better approach is to separate the back-end data access from the presentation layer, by creating two models – a *service provider model* (which accesses the back-end data and services) and a *service consumer model* (which provides the front-end user interface.) By using this approach, you can create a well-defined interface that separates the user interface from the back-end data. There are a number of valuable benefits to this architecture, including:

- **Reuse** - By encapsulating your back-end data and processes into a set of well-defined services, you can achieve high levels of reuse across your project.
- **Flexibility** – You can enable independence and flexibility between the presentation and the services layers, both at implementation and in deployment. For example, you can dynamically swap out the data service that drives a portlet user interface, without having to change the user interface.
- **More efficient division of labor** – By breaking development into two primary tasks – creating services and creating user interfaces – you can allocate your resources more efficiently. Some developers can focus on creating the data services, while others can create the user interface or service consumer models. Teams of developers can work independently on the layers without the need to constantly coordinate and adjust to modifications made on the other side.
- **Automatic service testing support –** When creating a service provider model, you can automatically generate a test harness, for easy testing of your back end functionality without requiring front end development.
- **Offline or disconnected development** – By leveraging the service builders, development can proceed without requiring access to the targeted back-end systems, which helps ease and speed development, especially for the user-interface developers.

Figure 6. Diagram showing the use of a service provider model to access data from a relational database, with separate portlet or consumer models



Creating a service provider model

This section details the steps required to build a service provider model.

Step 1: Create a new, empty (service provider) model.

Often, for clarity, it makes sense to save all of the data services for a project within one folder. For example, if you are creating a manufacturing portlet, you might save your service provider models in the folder `WEB-INF/models/manufacturing/data`.

Step 2: Add a data integration builder to your model.

The following table should help you determine which builder to use for which back-end system:

System or Data Source	Builder
Databases	SQL Call
Domino	Domino Data Access
Excel	Excel Import
Java class	Linked Java Object
PeopleSoft	PeopleSoft Component Interface

SAP	SAP Function Call
Siebel	Siebel Business Component
REST Service (Representational State Transfer)	REST Service Call builder
Web Service	Web Service Call Web Service Multiple Operations
XML Data	Variable or Import to XML
SAP Business Warehouse/MDX	SAP BW Data Access

For example, if you want to get data from a JDBC-compliant database, you need to add a SQL Call builder to your model, which lets you specify inputs such as your SQL statement. After you fill out the builder's inputs and save the builder, it will then generate a schema that describes the data, a variable typed with the schema, and an execute method that, when invoked, executes the SQL query and places the returning data in the variable.

When creating services, schemas play an important role. Not only do they describe the data being returned by the back-end system, but they are also heavily used by the builders within WebSphere Portlet Factory to generate the portlets' user interfaces. Builders like SQL Call, Domino Data Access, Excel Import, PeopleSoft Component Interface, SAP Function Call, and Siebel Business Component automatically generate the schemas for you.

Step 3: Add a Service Definition builder to your model.

This builder creates the underlying support for an initially empty service, provides an option for exposing the service through WSDL, and names the service. When adding the Service Definition builder, be sure to select **Add Testing Support** so the builder will auto-generate Web pages for testing the service and all the operations in the service.

Step 4: Add one or more Service Operation builders to your model.

This builder adds an operation or method to the service defined by the Service Definition builder. The operation can invoke any data access support available in the model. As described above, data access is usually provided by using one or more back-end integration builders, such as SAP Function or SQL Call, but can also be provided by a linked Java object or any data-returning method.

For example, if you are creating a service that gets orders data from a JDBC-compliant database, you first need to add a SQL Call builder, as described above. Then, you will add a Service Definition builder, giving the service a name (such as `OrdersData`). Finally, you will add a Service Operation builder, giving the operation a name (such as `getOrderDetails`), pointing at the execute method to wrap as a service, and specifying the input and result schemas.

Note: The Service Operation builder:

- Does not invoke the external data access API. Other builders (such as the SQL Call builder), make the calls and bring the data into the Web application.

- Can transform data from the structures used by the back-end system builder calls into the desired structures for the service being defined.

Step 5: Test the model by running it.

Use the auto-generated test harness to ensure that service is operational. Remember, you need to select the **Add testing support** option in the Service Definition builder in order for this to work. In addition, if your service takes inputs, you can set default input values in the Service Definition builder, by selecting **Specify Default Inputs** in the **Testing Support** section.

Step 6: Add additional operations to your service.

It is often useful to define a service with multiple, related operations. Sometimes the data sources can even come from more than one back-end system. For example, a customer information service can define operations to return a customer list (based on search criteria), customer detailed information, customer orders, customer complaints, and so on.

If you want to expose multiple operations in your service, then you will need to add additional data integration builders and service operation builders to your model for each operation that you want to include in your service. For the orders example described in Step 4, you might want to add another operation called `getOrdersbyCustomer`, which will return all of the orders for a given customer. To add this operation, you would add a SQL builder that defines the query (since the data in this example comes from a database) and then add another Service Operation builder that wraps this latest SQL call execute method as a service.

Step 7: Generate a stub model.

After your initial service is tested, you might want to generate a stub model, which is an entirely self-contained model that declares the same service, operations, schemas, and so on, as the original service, but which does not require back-end access. To generate a stub, simply provide a name for your stub model and click **Generate Stub** in the Service Definition builder.

Stub models are often very useful for team development projects, where there are some developers creating the back-end access and others who are creating the front-end for the portlets. For example, assume a user interface developer needs a service you are developing in order to start working on her portlet. You can generate a stub and give it to the user interface developer. This stub can easily be swapped out for the final service when it is complete. Stub models are also useful when your back-end system is slow in responding or has limited availability. By using stubs, you will save considerable time because the stub model does not need to access the slow back-end system for every regeneration cycle.

Additional features

This section provided a high-level overview of the most common steps used to create data services. Periodically, you will need to perform additional steps to create your data services, which may include transforming the data, performing some pre- or post-processing actions, and adding error handling. Common builders used for these types of tasks include Action List, Error Handler, and Linked Java Object builder.

Given the scope of this document, it cannot cover all of these steps, nor can it detail all of the service features provided by WebSphere Portlet Factory, which includes dynamic service mapping, service interface support, automatic generation of service documentation, and more. Refer to the WebSphere Portlet Factory help system for more details on these additional capabilities. There are also some samples showing how to create services for various back end data sources available at:
http://www-10.lotus.com/ldd/pfwiki.nsf

Developing the portlet user interface

This section describes the high-level steps for creating your portlet user interface.

Creating a service consumer model

If you are following best practices, the first step in creating your portlet user interface is to create a new service consumer model, to which you will add all of the builders necessary to call the appropriate data services and generate the portlet user interface.

The next step is to add a Service Consumer builder to this model, which is used to provide access to a public service created by the builders in a service provider model. The Service Consumer builder can be used to add all operations within the selected service provider to your model, or you can optionally select one operation to add to your model.

Note that you may add multiple service consumer builders to your model to access multiple service providers.

For an example of a service consumer model, navigate to the samples folder under `WEB-INF/models/samples/gettingstarted/OrdersServiceConsumer.model`. Note that you must first install the **Tutorials and Samples – Applications** feature set to access this sample.

Creating the initial presentation interface

After a service is available, the next step is to build the portlet user interface. The most common builder to use for this is View & Form. This builder is a high-level builder that automatically generates pages for viewing and editing data from data services.

View & Form builder

The View & Form builder can be used to create a portlet with a view page that displays the results from a service operation, with optional pages for input, details, and update. As with other builders, you can selectively add the functionality that you require. For example, you can have this builder just create the view page and not the input, details, and update pages.

Here are a few examples of portlet use cases ideally suited to the View & Form builder:

- A portlet displays a list of expenses, and allows users to click to see the details of an expense, and then update that expense.
- An employee directory portlet enables users to search for an employee and view a table of results or the selected employee.
- A “Personal Information” portlet in an HR employee self-service application allows users to view and edit their own personal data.

The View & Form builder automates the creation of:

- A view page that displays a table or single record of data.
- An optional input page for the specified view operation, with optional validation.
- An optional detail page that displays either data from the selected row in the view page or data from another data service or method.
- An optional update page for editing details or results data.
- Navigation between the pages (buttons).

Alternative methods for creating the initial presentation

In some cases you may need to create the initial presentation for a portlet using a combination of other builders instead of using View & Form. See section below on "Creating the initial portlet user interface."

Adding builders to achieve the exact functionality you need

After you have created your initial presentation page or pages, the next step is usually to add additional modifier builders that add new features to your portlet. For example, if you use View & Form builder to create a page with a table of service results, you can then add a Data Column Modifier builder to rearrange or hide columns, or to add column sorting functionality. Or, if you want to add a navigation link, you can use a Link builder.

The development process then continues by iteratively making changes to the model and testing them out. At any time you can open any of the builders in the model and change any of the values. You can also temporarily disable any builders, by using the right-click menu on the builder in the Outline view.

As you continue to customize your application by adding and editing builders, the number of builders in the model will increase. As a best practice it is recommended to keep the number of builders in a model under 50, since models larger than that tend to be harder to work with. In order to keep the number of builders small, you can use builders such as Linked Model and Model Container to make your application more modular.

Selecting builders for common tasks

This section lists common tasks and describes the builders that are typically used to accomplish each task. Note that this section does not cover all of the builders contained in WebSphere Portlet Factory. Rather, it covers the most commonly used builders.

Creating services

Task	Builder
I want to get **access to a data store.**	Use one of the following data integration builders: • **SQL Call** (relational databases) • **Domino Data Access** (Lotus Domino) • **Excel Import** • **Linked Java Object** (Java class) • **PeopleSoft Component Interface** • **SAP Function Call** • **Siebel Business Component** • **Web Service Call** or Web Service Multiple Operations (any existing web service) • **REST Service Call** (any existing REST service) • **Variable** or **Import to XML** (XML file or data) • **SAP BW Data Access** (SAP Business Warehouse MDX)
I want to **create a service** with one or more operations. I may use this service from another WebSphere Portlet Factory model using lightweight Java execution, or I may make it available to other applications via WSDL and SOAP.	**Service Definition** and **Service Operation** Define the overall service settings with Service Definition, then add one or more operations with Service Operation. Select "Generate WSDL" from the Service Definition builder to make the service available via WSDL.
I want to **generate a service test harness** for testing a service.	**Service Definition** Enable the "Add Testing Support" input, in the Testing Support section. When you run the service model, you will get an index page with a link for each service operation. You can provide default values for service inputs by selecting "Specify Default Inputs" in the Testing Support section.
I want to **create a stub service model**, so that I can create a presentation model using my service interface but without needing the actual back end implementation.	**Service Definition** Enter the desired name of the stub model in the "Stub Model" input, and then click the "Generate Stub" button.

I want to **switch between service provider implementations** without having to edit my consumer models. For example, I want to automatically use a stub implementation of a service.	Use the **Service Mapping Registry** features of the Service Consumer builder. Service mapping is controlled with the XML files in the WEB-INF/config/service_mappings folder of a project. For more information, search in Help Contents of the product for "Service Mapping Registry".
I want to make a service available to other applications using **WSDL and SOAP.**	**Service Definition** Enable the "Generate WSDL" input. To get the URL of the WSDL, enable the testing support and run the model. The test index page has a link to the WSDL URL.
I want to **transform data** between a schema used by my back end data and another schema.	Use the **Service Operation** builder for simple mapping. For more complex XML manipulation, use Java code that manipulates IXml, accessing the Java code with the **Linked Java Object** builder.

Creating the initial portlet user interface

Task	Builder
I want to **access a service** created in a service provider model.	**Service Consumer** This builder makes a service available for use. The service must be defined in a service provider model, which is a model that uses Service Definition and Service Operation to create a service. You can also access a web service using **Web Service Call**, but the recommended best practice is to put the web service access in a provider model so that you can use features such as the automatic test harness and the stub service model support.
I want to create **view and form pages** based on data from a data service. The view page will display a **table or single record of data** from the service results. I may also want to create associated pages such as: • An **input page** for the specified operation, with optional validation. • A **detail page** that displays either data from the selected row in the view page or data from another data service or method. • An **update page** for editing details or results data.	**View & Form** As described above, the View & Form builder can be used to generate several common UI page patterns.

I want to implement a **details page** triggered by clicking on a row in a table.	Use the **View & Form** builder to create your table and select the choice for "Create link to details". You then have three choices for how to handle the link action: • The details page can show the data already retrieved in that table row. • The details page can show data from another operation, such as a service operation that retrieves details data. In this case, you will need to use the Input Overrides feature of the **Service Consumer** builder to specify the service inputs needed to retrieve the details data. These inputs can typically come from the SelectedRowData variable created by the View & Form builder. • The link can be used to execute any action you specify, such as an Action List or a page. With any of these link types, the View & Form builder automatically creates a <name>_SelectedRowData variable that is populated with all the contents of the row that was clicked. Typically you will select values from this variable when choosing the inputs you need for your row action.
I want to create an **input page** for my data service operation, which **does not return result data**.	**Input Form** Creates an input page for a data service operation or a method. It is much like View & Form input page support, but the next action after submitting the input form is a user-specified action. This builder is suitable for operations that do not have result data to display.
I have some **existing HTML (or JSP) that I want to use** in my portlet and add data or other elements to.	**Imported Page** Adds an existing HTML or JSP page to the current Model. By using Imported Page builders, you can develop your HTML or JSP code in your favorite editor and then import the results into the Designer. After you import a page into the Designer, you can use builders to modify those pages.
I want to **create a chart**.	**Web Charts** The Web Charts builder adds a chart to a page at a specified location. The data for the chart can come from a variable, a service operation, or any other data source in the model.

I want to use **Workplace Forms** in my portlet.	Use the **Lotus Forms Embed** builder to embed a Workplace Form on a page. The form is retrieved from the forms repository and rendered to the user using the Workplace Forms Viewer. Use the **Lotus Forms Launch** builder to add a control (button or link) to a page in your model to open a Workplace Form. The form is opened in a separate browser window.
I want to **add some structured data to a page for input or viewing** and have fields automatically generated based on the schema of the data.	**Data Page** Creates a structure based on the XML data in a variable and renders this structure as a form (both input and display) on a page. The structure created by the data page describes the data to be displayed as well as how to render the data on the form. A schema associated with the data on which the data page operates can determine how the data gets displayed as can the HTML page associated with the data page. Note that the Data Page builder is used under the covers by high-level builders such as View & Form.
I want to quickly **create a simple HTML page** for use with other WebSphere Portlet Factory builders.	**Page** Used to create a simple user interface using HTML or JSP markup. You can enter or paste markup directly into the builder.
I want to create **view and form pages** based on data from a **specific back end system**.	**Domino View & Form, SAP View & Form, PeopleSoft View & Form, Siebel View & Form** These are back end specific builders that can be used to create the initial user interface for a portlet. However, with these builders you don't get the benefits of a service provider/consumer architecture.
I want to use **Flash** in my portlet.	**Flash** Displays an Adobe Flash SWF file at a specified page location.

Working with tables, columns, and data layout

Task	Builder
I want to clean up my portlet by **hiding columns** that do not need to be displayed.	**Rich Data Definition (RDD)** RDD is a powerful builder that you will use in most, if not all, of your Models. This builder lets you select a schema in your Model. For each field in the selected schema, you can specify labels, control types, lookup tables, formatting, validation, and much more – just by using a single instance of this builder. Alternatively, you can use the **Data Column Modifier** builder which is described below.
I want to add **sorting** to the columns in my table.	There are a few different ways of adding sorting: **Rich Data Definition –** For each of your columns that you want to add sorting to, select the associated element in the RDD builder, and select one of the base data definitions that specify sorting. **OR** **Data Column Modifier** – This builder allows you to modify table columns, including adding sorting.
I want to: **Rearrange** the order of the columns in a table **Add a counter column** to my table Change the **alignment** of my columns Add an "**Edit" column** for editing records in your table Add a **"Delete" column**	**Data Column Modifier** This builder controls the tabular display of one of the "container" elements in the data being managed by a Data Page builder call. When you choose one of the available elements, the builder editor displays a table that lists all the child elements in the selected element. This table contains inputs allowing you to configure the display characteristics of each column in the table as well as adding columns and rearranging their placement.
I want to create a data view that allows users **to expand and collapse the data by category**. This type of control is often referred to as a "twisty" by Domino users.	**Category View** This builder adds collapsible sections associated with one or more columns of categorized or sorted data. A user can click on an icon to expand and collapse displayed category data. Note that this builder requires that a column of the data contains category values.

I want to allow users **to expand and collapse data at a specific location**.	**Collapsible Section** This builder arranges for an additional information section to be displayed (or hidden) when the user clicks a specified area on the page.
I want **to create visual grouping of data fields** for display in forms.	**Data Hierarchy Modifier** This builder lets you select data from a builder such as View & Form or Data Page, and create new grouping within the data structure.
I want **to create newspaper-style columns** for form data.	**Form Layout** This builder lets you select form data for a page, and divide it into multiple vertical columns.

Working with variables and schemas

Task	Builder
I want to **create an XML variable**.	The **Variable** builder let you type or paste XML data; the **Import To XML** builder imports an XML file.
I have an **existing schema file (.xsd)** that I want to use in my Model.	**Schema** Imports a schema for use in your Model.
I want to **create a simple schema.**	Use **Variable** builder to create an XML variable with the desired structure, and then use **Simple Schema Generator** to make a schema from that Variable.
I have a schema, but I do not have any **sample data** to go along with it.	A nice feature of the variable builder is that it can generate some sample data for you. Simple select the **Variable** builder and use the Type select to navigate to your schema. After you have selected the schema, a “Create Sample Data” button will appear. Click this to generate your sample data.
I have an **existing Java Bean** that I want to use in my application.	**Linked Java Object**. Once you have used Linked Java Object to make the Bean available in your model, you can use the Data Page builder to generate page fields directly from Java Bean properties. **Java/XML Converter** can also be used to convert beans to XML.

I need to **transform data** between schemas.	Use the **Service Operation** builder for simple mapping. This is usually done in a service provider model. For more complex XML manipulation, use Java code that manipulates IXml. See the Javadoc for the IXml class for details.
I want to **share some data across portlets**. For example, I may want to create a "filter" portlet that lets users specify values that are used by multiple other portlets on the same or different portal page.	**Shared Variable** This builder marks a variable so that it can be shared across portlets and models.
I want to **reduce the session size** for my application.	**Discardable Variable** This builder modifies an existing variable so that it is not stored in session.

Controlling the validation, formatting, labels, and behavior of data fields

Task	Builder
I want to **add formatting or validation** to a field or to an element in a schema. For example, I want to: **Format** the field (for example, format the data in the acceptable date format) **Translate** an expression (for example, remove dashes from a formatted string to that it can be properly saved) **Validate** the field being submitted (for example, ensure that the Birthday field is a date).	Although either **Data Field Modifier (DFM) or Rich Data Definition (RDD)** can help with these tasks, RDD is the best choice because you can use this builder to encode and reuse detailed field behaviors. You can enable reuse at the schema level by creating an RDD file for a particular schema, and you can also reuse a single shared library of common field types, with the "base" data definition functionality. To generate your own data definition file, use the RDD's builder interface to set the appropriate formatting, validation, and so on. Then, scroll down to the bottom of the builder and press the "Create Data Definition File" button.

I want to make my **column names easier to read.** For example, rather than displaying CUSTOMER_NAME, I want to just display the word Customer.	The **Data Column Modifier** and **Rich Data Definition** builders can be used to change the labels that are displayed in your portlet. The **Localized Resource** builder can also be used to change column and field labels, and it can support multiple languages. It lets you pick a resource bundle, which it imports into a variable in your Model. You can then use the values from that variable to localize text strings in your model, such as button or column labels. After you have added the Localized Resource builder to your model, the next step is to open up the builder that created the fields (View & Form builder or some other builder based on Data Page), and scroll down to find the Label Translation section. From there, select the locale data or resource bundle you are using.
I want to **change the behavior of a field** in my view or my form. I want to: Make the field a data entry control (such as a text input, text area, select, or check box). Make the field view only (for example, text or image). Make the field an action field (such as a link, button, or image button).	Use **Data Field Modifier** to change the behavior of fields created by Data Page, or any higher-level builder that uses Data Page (for example, View and Form). **Rich Data Definition** can also be used for much of the functionality provided by Data Field Modifier. Note that some of the features of Rich Data Definition are available in the XML files used by RDD and are not available in the RDD builder user interface.

I have a **value that I want to translate into a meaningful name**. For example, instead of displaying the Department Code, I want to display the Department Name. I have an edit form that allows users to select a name (like Department Name), but I really need to save the associated value (like Department Code). I want to populate my select list (or similar control) with a value that is not returned from my database, such as a blank or an option such as "any."	The most common way to do this is by adding a **Lookup Table builder** to your Model. The Lookup Table builder creates a lookup table, which is used to translate between some computer-readable ID and a human-readable name, often the result of a database query. The table also caches the results across different users, including the opportunity to prevent data from becoming stale. This builder: • Lets you access a data source by entering a SQL statement, pointing to an XML variable, or typing in some XML. • Adds to the query results any name/value pairs you want to include in the select list (for example, blank). • Creates methods that do the translation between name/value pairs. Next, add the relevant page control builder to your Model, pointing it at the Lookup Table builder. Builders that work with Lookup Table include Data Field Modifier, Select, Text, and Radio Button Group. You can also have the **Rich Data Definition builder (RDD)** automatically create and apply a lookup table. This can be especially useful if you use the same lookup numerous times. See the help for RDD for more information
One of the builders I used in my Model generated a label that I do not like. I want to shut off **label generation**. OR I want to **provide my own label** for the field.	**Data Field Modifier** Data Field Modifier is a builder that is useful for a number of different tasks. Basically, this builder can be used to control the page elements created by Data Page builder (as well as the higher-level builders that leverage Data Page). This builder lets you select one or more fields and one or more data types (like double, string, and so on). You can then perform modifications including hiding, formatting, labeling, and determining the display of child elements. **Data Column Modifier and Rich Data Definition** These builders can be used to specify field labels.

Using actions and events

Task	Builders
I want to perform some **processing** in my Model such as: **Calling a method** and then returning a page. **Executing several back-end cal**ls before loading the initial page in my Model.	**Action List** The Action List builder allows you to add a set of one or more actions to your WebApp. Typically, all Models must have an action in them called **main**. Most of the higher-level builders automatically create this action for you. However, this builder includes the ability for you to override this feature, so that you can add your own main action list to your Model, which can perform whatever processing you need.
I want to perform a **conditional statement** (for example, if two variables contain the same value, perform an action), but I do not want to write Java code to do this.	Within the **Action List** builder, you can build conditional statements without having to write Java code. Use the picker underneath the actions user input to navigate to Special -> Conditional. From there, you can select if, else, endif, and build up statements that will execute an action or return a page based upon conditional logic.
I want to **debug** by sending SystemOut messages to the console.	Within the **Action List** builder, select the SystemOut option underneath “Special” within the action picker. This builder sends SystemOut messages to the application server console, and to the debugTracing log located in the deployed .../WEB-INF/logs directory. For example, in WebSphere Portal, the log files for running as a portlet are found in the deployed portlet WAR's directory. If the portlet is run standalone, log files can be found in the deployed enterprise application's directory.
I want to **assign or append values to variables.** In other words, I want to append or replace what is in a given variable with the results of a method call, an argument, or another XML variable.	Within the **Action List** builder, select the Assignment option underneath “Special” within the action picker.

I want to use some **Java code** in my Model.	**Linked Java Object** This builder lets you point at a Java class. It then adds the public methods of that class to your WebApp, so you can call them from any builder input where you specify an action. In some scenarios, you could also use the **Method** builder**.** This builder lets you easily create a simple Method, which can make calls to display pages, call methods, or execute service calls. Note that if you need significant amounts of hand-coded Java, you should write a Java class and bring it into the Model with a Linked Java Object builder instead of using Method builders. This will make the Model easier to manage, and the code assistance is better in the Eclipse Java editor. In general, Method builders in your Model should only contain small amounts of "glue" code.
I want to add **error handling** to my Model.	**Error Handler** This builder catches Java exceptions that may occur within the run-time execution of a Web application's actions. The error handling can be specified on any action, such as a page, method or action list. You can also set a general handler for the entire application to be used if an action has no error handler. This type of error handling is modeled closely after the Java try-catch exception handling and JSP page directive, where each page can set the error page at design time. In all of these cases the lowest-level handler (catch) is called to handle the exception.
I want to add **caching** to my Model. For example, I am populating an Employee select list from a database call. Since the list of employees does not change that frequently, I want to cache the content when I have received it from the database.	**Cache Control** This builder caches the output of a specific action within a Model. Note that this caching is across users: all users in a given profile who access that Model will be presented with the same cached data. When this caching is suitable, it can substantially improve performance. **Service Operation** and **Lookup Table** builders also have built-in caching options (this is implemented under the covers with Cache Control).

I want to establish **communication between portlets** using simple broadcast events.	**Event Declaration** **Event Handler** Use Event Declaration to define the event name and any arguments. Use the same declaration in both models (a convenient way to do this is to put the declaration in a separate model and then to use Imported Model to bring it into both models). In the source model, call the "fire<EventName>" method when you want to send the event. In the target model, use the Event Handler builder to specify the action you want to perform when the event is received.
I want to establish **communication between portlets** using **portlet wiring** or **Click-to-Action.**	**Cooperative Portlet Source** **Cooperative Portlet Target** Use Cooperative Portlet Source in the source model. Also in the source model, call the "fire<EventName>" method when you want to send the event.. In the target model, use the Cooperative Portlet Target builder, and then use the Event Handler builder to specify the action you want to perform when the event is received. The Output information in Cooperative Portlet Source must match the Input information specified in the Cooperative Portlet Target Builder.

Adding navigation and page actions

Task	Builder
I want to add an **action control** to a page, such as a **Button, Link** or a **clickable image.**	**Button** **Image Button** **Link** These builders can be used to add action controls to a specified page location. The action can be a model action or an external URL. Note that if the action is part of a form, you should select "Submit form and invoke action" as the Action Type.
I want to specify a **form action** to run whenever a form is submitted.	**Form Submit Action** This builder attaches an action to an HTML form. Specify the form's location as the Page Location input. When you use View & Form builder to create a form, this builder is automatically used under the covers.

I want to **run an action when a user clicks or types** in the portlet. For example, when a user selects a customer from a select list, I want to run an action (for example, search a database and return a table of orders that belong to the selected customer).	**HTML Event Action** This builder allows you to associate a function with an HTML behavioral attribute. The HTML behavioral attributes are those that correspond to HTML events, such as OnClick or OnChange for INPUT tags. The HTML Event Action builder can only define actions for the events that a particular tag supports. To build the example described in the task on the left, you would likely use the following builders: **Service Consumer, View & Form, Data Field Modifier, Lookup Table**, **Select** and **Variable** to build the select list and page. **HTML Event Action** builder, to call an action (created by an Action List builder) on the onChange event. This action would get the appropriate orders from the database and return the page.
I want to **generate a pop-up menu** with options when a user clicks in a certain part of the page.	**Contextual Menu** Adds a pop-up menu and its menu items pop up when users click a specific link, button, or icon.
I want to provide a **link or button for downloading or launching a file**.	**Content Launch Action** Adds a button or link to a page that opens a file or resource that might require an additional plug-in to view or handle the content. For example, if you want to launch a spreadsheet or video file from a page.
I want to create a link in my portlet that **links to a different portal page**.	**WebSphere Portal Link** With this builder, you can link to another portlet or page in WebSphere Portal. The object you link to is referenced with a WebSphere Portal resource identifier. See the builder help for more information.
I want to create a portlet that includes **tabs** to navigate between the various pages in my portlet.	**Page Tabs** This builder adds tab navigation to a set of pages in a model. The tabs are added to all the specified pages.

I want to add **paging controls** to my portlet, for paging through a large result data set.	Some of the higher-level builders like View & Form include built-in support for paging. If you are not using one of these builders, then the best approach for adding paging is to use the **Data Field Modifier** or **Data Column Modifier** builder. Each of these builders have inputs specific to paging. Note that the Paging Assistant builder operates at a low level (and is actually called by the Data Field Modifier and Column Modifier builders). When invoked directly, the Paging Assistant may not correctly handle paging in all possible configurations of Data Page.
I created a **form** that accepts user input. When the user clicks **submit**, I want to display the results, but the results page does not seem to display – even though I am linking to the right action.	Make sure to use "submit form" actions when updating server state and "link" events for navigation. For instance, if you add a page action builder (Button, Link, Image Button, and so on) as the method of navigating to the results page after an input form is submitted, make sure you set the Action Type as “Submit form and invoke action” or “Submit form and invoke URL”. For page-to-page navigation set the action type to be “Link to Action” or “Link to URL”.

Adding and controlling page elements

Task	Builder
I want to **show or hide a page element based upon certain conditions** or based upon a value. For example, I have a select list that allows a user to select a product. When a product is selected, all of the part numbers that make up that product are displayed in a table below the select list. When no product is selected, the table does not appear.	**Visibility Setter** Use the Visibility Setter to control what is displayed in your portlet. When you add a Visibility Setter builder to your Model, there are several ways you can control the visibility of the user interface element the Visibility Setter is attached to. You can: Hide a control always Hide a control based on a value Hide a control based on the comparison of two values In the example described in the associated task, the table is being hidden by a visibility setter. If the value in a variable is blank, then the table is hidden. When the variable contains a value, the table will be displayed.

I want to **change an HTML attribute**, but the builder I am using does not allow me to modify or add an attribute.	**Attribute Setter** This builder modifies existing elements on a page by replacing or adding to any HTML attribute values that those elements support. Use the Attribute Setter to add or modify attribute values not set by the builders in your Model. You can use this builder to change the font, size, border, or any other supported attribute for a control.
I want to **include the same page** in several of my portlets. Or I want to include **a page generated by a builder onto an existing page**. For example, when a customer is selected from a select list, I want to display a table below the select list. Or I want to **insert the display of a page in another model into a page in my primary model**.	**Inserted Page** Use this builder to insert the contents of a Page in the same model into a named location on a selected page (or pages). For example, you might want to insert a page created by View & Form onto an imported page. **Model Container** Use this builder to visually insert another model into a page. The other model can have its own page-to-page navigation without affecting the outer (container) model.
I want to display some **read-only text** on a page.	**Text** This builder displays some text from a Variable or any other data source at a specified page location.
Add some **client-side JavaScript** to a page.	**Client JavaScript** This builder can be used to add JavaScript code to a page. **HTML Event Action** This builder can be used to invoke JavaScript based on a client HTML event such as onClick, onChange, or onBlur.
I have an existing **tag library** I want to leverage in my portlet.	**JSP Tag** This builder automates the process of using Java Server Page tags from a tag library. After you select a tag library descriptor file in the builder's editor, you can quickly select a JSP tag to use on the page. After a tag is selected, the builder displays the tag's attributes, so you can easily specify the inputs.

Using Ajax and related techniques

Task	Builder
I want to have **just one part of a page refresh** when the user does an action.	There are two primary ways you can do this: 1. The page action builders (such as **Link**, **Button,** and **HTML Event Action**) have an option for **Post-Action Behavior** that can trigger a partial-page refresh instead of refreshing the entire page. The builders with this support include Link, Button, Image Button, HTML Event Action, and Form Submit Action. 2. The **Ajax Region** builder can be used to mark an area of the page, an entire page, or an entire model for partial refresh. With this builder, any actions triggered from within the selected area will cause a refresh of that area only.
I want to create a **popup window** without the containing portal or other surrounding pages.	Use a page action builder such as **Button, Link, HTML Event Action** to invoke the page or action for the popup. Then in the **Post-Action Behavior** section, select "Show Advanced Options" and choose "Show action results standalone: display no containing pages".
I want my portlet to support **drag and drop**.	**Dojo Drag Source** **Dojo Drop Target** Together, these two builders provide drag and drop features on a page. Use the Dojo Drag Source builder to designate a location on a page as the drag source and to specify a single piece of data to be passed to the target action. Use the Dojo Drop Target builder to designate a location on the page as the drop area and to specify a method action to invoke when the drop occurs. The method will be passed the data from the drag source. The drag drop can operate on a single page or span across multiple pages, such as in a portal environment.

I want to support **dynamic editing** of the tabular data my portlet displays.	**Dojo Inline Edit Builder** The Dojo Inline Edit Builder allows you to add editing capabilities to a Data Page field so that it initially appears as read-only text. The text can be edited by clicking a pencil edit icon next to the text field. Once editing is complete, the user has the option to cancel the changes or save the results either to the page (as a hidden input) or submit the outer form.
I want a text input field with **Ajax-based type-ahead** so that when a user types, they immediately see a drop-down list of suggestions.	**Ajax Type-Ahead** Attaches to an HTML text input field, and provides what is generally known as "type ahead" functionality for that input. The list of choices for the drop-down can come from any XML in the model or from a Lookup Table.
I want to **validate** a **form before** the **user submits** it.	**Dynamic Validation** Used to validate a user's input to fields on a form as the user is working with the form. This builder enables incremental client/server validation. It validates the form data and displays any error messages whenever an input value is edited.
I want to **display a tooltip** when the user hovers over a certain part of the page.	**Dojo Tooltip Builder** The Dojo Tooltip Builder allows the application to display a popup text box when the user's mouse hovers over a specified target on a page. The tooltip text can be a simple caption string, or an entire page in your model.
I want the **row** the **user hovers over** to be **highlighted** in a different color.	**Highlighter** Highlights a row in a table (or a field in a form) as the user's mouse moves over it.
I want to use **client-side events** between portlets or within a portlet.	**Event Declaration** **Client Event Handler** In the Event Declaration builder, select "Use client side delivery". Then use the Client Event Handler builder to handle the event that has been marked for client-side delivery.

I want to **access data – HTML, XML, or JSON – for use in client JavaScript.**	You can generate a standalone URL to any model page or action by using the **WebAppAccess method getBackChannelUrl**. To automatically generate JSON (JavaScript Object Notation) data from XML data, you can use the **XML/JavaScript Converter builder**. The JSON data can then be consumed by existing or custom UI elements on the client. This builder can be used in conjunction with the **Link**, **Button**, **Image Button**, **HTML Event Action**, and **Form Submit Action** builders, when Evaluate action results as JavaScript is set as the Post-Action Behavior in those builders.

Integrating WebSphere Portal features

Task	Builder
I want to **add portlet-to-portlet communication** such as **Click-To-Action** or **Portlet Wiring.**	See above under Using actions and events.
I want to integrate **"people awareness"** or SameTime chat capabilities into my portlet.	**People Awareness** This builder can make names "people-aware." It interacts with the Lotus® Collaborative Components available within Portal.
I want to **link to another portal page** from my portlet.	**WebSphere Portal Link** With this builder, you can link to another portlet or page in WebSphere Portal. The object you link to is referenced with a WebSphere Portal resource identifier. See the builder help for more information.
I want to add custom **edit or configure mode** options for business users or administrators.	To enable Edit or Configure support, you must first profile the builder inputs that you want to make available for runtime configuration. Then specify the desired Edit/Configure behavior in Portal in the **Portlet Adapter** builder. To create a custom UI for Edit or Configure, you must then create a customizer model that includes the **Portlet Customizer** builder. In the Portlet Adapter builder of the portlet model, specify your customizer model as the "custom model" choice.

I want to have **different variations** of my application for different **Portal groups or roles**.	There are a few steps in doing this: - Profile the builder inputs that you want to vary. - Create a named profile corresponding to each of the Portal groups or roles, and set the profile values as desired for each profile. - In the Profile Set editor, in the Select Handler tab, choose “WPS Group Segment Handler” as the Profile Selection Handler. - From the Manage Profiles tab of the Profile Set Editor, in the Advanced section of the Edit Profile dialog, select Add External and enter the name of the Portal group, Do this for each profile. - In the Portlet Adapter builder for your model, select “Do not expose in Portal tools” for your profile set. For more information, see the “Profile portlets” article available from the “Miscellaneous Techniques” link on the product documentation site referenced at the end of this article.
I have a model that I want to make **available as a portlet.**	**Portlet Adapter** Any model containing the Portlet Adapter builder becomes available as a portlet. When you first portlet-enable a model this way, you must save the model file, then use the Rebuild Portlet WAR command, to update the portlet WAR in WebSphere Portal.
I want to implement **single sign-on** by storing user credentials in WebSphere Portal’s credential vault.	**WPS Credential** This builder provides methods for saving and retrieving credentials (user names and passwords) from Portal’s credential vault.
I want my portlet to support **searching for people or groups** in WebSphere Portal.	**Directory Search** Provides a way to search for people or groups in WebSphere Portal or other directories.
I want my portlet to **display content from** IBM Workplace **Web Content Management**.	**Lotus Web Content Management Access** Allows you to retrieve content and components from IBM Workplace Web Content Management.

Sharing functionality or resources across models

Task	Builder
There are **common builders that I need to use in all of my Models**. How do I architect this?	**Imported Model** Use this builder to import the complete builder list of a second model into your model. In the simple case, you can import all the builders as is. If you want to parameterize builder inputs in the second model, you can profile those inputs and set the Profile Handling input to "Set individual inputs".
I want to **reuse one or more pages from a model within another model**. How do I do this?	**Model Container** Use this builder to visually insert another model into a page. The other model can have its own page-to-page navigation without affecting the outer (container) model.
I want to **reuse the actions or methods from a model within another model**. How do I do this?	**Linked Model** Use the Linked Model builder to provide access to any method, Linked Java Object (LJO), or action list in another model. You can also access pages in the linked model, but in general Model Container is better for reusing visual elements of another model.
I want to have **centralized automated control of all my field behavior** for each field types. For example, I want all my date fields to use a calendar picker with formatting and validation that I can control. Or, I want all my tax ID fields to use formatting based on regular expressions. Or, I want every input for a US state to use a drop-down list. Or, I want every model that uses a particular schema to automatically use a standard set of behaviors defined for that schema.	**Rich Data Definition** This builder can be used to automate all the UI behavior for fields that are generated from schemas.
I want **define a common look and feel for pages** in all the models in my project.	See the section on Customizing the user interface and styles for information on how to use custom HTML layout pages, HTML templates, and style sheets to control the look and feel of all the pages in your application.

I want to **automate a common design pattern** that I use in a number of models.	You may want to consider creating a custom builder. See the WebSphere Portlet Factory product documentation for more information.
There are a number of **builder inputs in multiple models that I always want to have the same value**, and I want to be able to edit that value in a single place. For example, I have SQL Call builders in multiple models, and I want to be able to easily edit the SQL DataSource name for all the builders and models.	Profile all those builder inputs to the same profile set and the same profile entry. Then whenever you edit the value, all the builder inputs in all the models will be updated automatically.

Supporting translation and localized strings

Task	Builder
I need to support **multiple languages** for a portlet**.**	There are a few steps involved in creating a multi-language portlet: • Put text strings that should be localized into resource bundle (.properties) files. Some builders such as Data Page have support for automatically generating a properties file for a set of field labels. • Use **Localized Resource** as one of the first builders in your model and specify your resource bundle. • In builders such as **View & Form** and **Input Form**, select your resource bundle for "Resource Bundle Name". In some other builders such as **Data Page** you can reference the LocaleData variable created by Localized Resource. • Profile the Language and Country inputs of the Localized Resource builder to the LanguageCode and CountryCode entries of the com.bowstreet.profileset.SimpleLocaleValues profile set. See the sample article for localizing portlets on the product documentation samples web site (see resources at end of this paper).

Customizing the user interface and styles

Many times, developers are satisfied with the user interface generated by WebSphere Portlet Factory. Other times, developers would like to modify what the product generated – whether it is the style of the chart or the design of the portlet user interface. The following details the most common ways of achieving the style that you desire:

Supply your own style sheets or HTML layout pages.

The builders provided by WebSphere Portlet Factory include default HTML layout pages and style sheets (which provide styles for elements such as tables, header cells, and alternating row colors). By changing these files, you can easily control the design of your portlets.

Note that the HTML layout pages that ship with the product contain named tags that indicate where key controls should be placed. If you want to move the control to a different place on the page, simply move the named tag within the HTML file. If you supply your own HTML file and you discover that a control is not appearing (but used to appear with the default HTML page), you probably do not have the right tag name in your file.

For example, the View & Form builder uses a default layout page with the following named tags:

```
<span name="search_section" />
<table name="data" cellspacing="0"
cellpadding="3" />

<span name="paging_buttons" />
<br>
<span name="back_button" />
<span name="update_button" />
```

You can create your own copy of this layout page and add any other HTML you want on your pages. For example, if you want additional static text or graphics around the data area, you can add it here by putting it before and after the table named “data”. Then you can specify your page in the View & Form builder, and that builder will insert the JSP code at the specified name tag locations.

Typically, developers will only make small modifications (if any) to the HTML files. They will create, for example, one default page from a builder for all of their portlets by copying and pasting the default page that comes with the product. This same page will then be used each time a developer on the team adds the builder to a model. Note that in this scenario, the developers are using the page generation capabilities in WebSphere Portlet Factory to generate much of the user interface.

Every so often, developers want to have pixel-level control over the placement of each field and control on the page (to the extent that browsers allow), and cannot achieve the desired look with the default HTML pages, HTML templates (described below), and the combination of modifier builders (like Rich Data Definition, and so on). In these rare scenarios, developers can create their own custom HTML pages that contain tags for each field that needs to be displayed.

For example, suppose you want to create a page that displays order details, such as **Order Number** and **Amount**. The most common way to do this is to leverage the page automation capabilities of the tool to generate the tags (such as order_number) into the HTML page. If you do not want to leverage the page generation capabilities of the product, you can supply your own HTML that contains all of the tag names (like order_number) that you want to see on the page. Note that the tag names must match the names within the associated schema. Also note that this approach hard-codes tag names into your page. If the underlying data or schema changes, you must update your HTML to reflect the new tags.

Modify or change the default HTML templates.

In the higher-level builders like Data Page, you will notice inputs for HTML templates. An HTML template provides instructions to the page automation builders and allows you to control the automatic generation of HTML.

Most of the HTML generated for portlets is a result of data page templates. These templates describe how elements like tables and forms should be generated. The templates create elements that reference styles defined in the default style sheet mentioned above.

You can use one HTML template for all pages in an application to achieve a consistent design or you may use different templates for different pages or sets of pages. HTML templates are used in conjunction with imported pages (Imported Page builder) which provide content layout, graphics and non-generated artifacts. They may also be used with style sheets as desired.

Given some of the intricacies of the template technology, most developers tend to stick with the default templates that come with WebSphere Portlet Factory. Or, they swap these out with some of the other templates shipped with the product.

For more information about HTML templates, refer to the WebSphere Portlet Factory documentation.

Debugging Tips

Here are some techniques that may be useful when debugging your applications.

1. Look for builder errors and warning messages in Designer. If there are errors shown in the "Problems" view of Designer when you are editing a WebSphere Portlet Factory model, you should track these down and resolve them, since these errors will likely result in execution errors.

2. Disable builders to temporarily simplify a model and isolate the problem. As in any development environment, simplifying an application to isolate a problem can make debugging easier. In WebSphere Portlet Factory Designer, you can disable builders by selected them in the Outline view and then right-clicking one of the selected builders and choosing "Disable".

3. Write data to system output to look at values during execution. There are a couple of ways to do this in a model:

- In an **Action List**, you can use the SystemOut action in the Special category to output values. For WebSphere Portal, this system output is written to the Portal server's log/SystemOut.log file.
- With the **Debug Tracing** builder, you can see the value of some data at a particular point during model execution. Specify the "Action to trace", and for "Additional debug output", select the value you want to see. This output is written to the WEB-INF/logs/debugTracing.txt file of the deployed WAR.

4. Look for errors in the event.log file of the deployed WAR. When a runtime error is detected during model execution, the details of the error will be logged in the file WEB-INF/logs/event.log file. The exception details may help you determine the root cause of the error.

5. Run with system tracing to see exactly what actions get executed. System tracing can be used to see all the high-level model actions during execution. System tracing also gives you a quick look at the performance of all the actions, so you can often quickly spot a performance issue. You can enable system tracing in the "Run" configuration dialog for WebSphere Portlet Factory Models, on the "Tracing" tab. The system tracing output is written to the system output file (for Portal Server, this is the server's log/SystemOut.log file).

6. Use Java-level debugging. To set up Java debugging (with single-step, breakpoints, etc.), you can use the remote debugging capabilities of Eclipse. See information in WebSphere Portlet Factory product help for how to set this up.

Deployment

This section covers at a high level a couple of important tips and best practices for deployment. For a more complete document on this subject, refer to the WebSphere Portlet Factory help system. A helpful document on project configuration and WAR deployment is located underneath **Working with WebSphere Portlet Factory➔ Overview: Deployment ➔ Deployment Configurations in a nutshell**. Another useful document about deploying portlet wars is located underneath **Tasks ➔Working with WebSphere Portlet Factory➔Overview: Deployment➔ Deployment for Production**.

Hint 1. Remember to add a Portlet Adapter builder to your model.

The Portlet Adapter builder tells WebSphere Portlet Factory to add your model to the WAR file. If your model does not have a Portlet Adapter builder in it, it will not be contained in your WAR file.

Hint 2. Change deployment properties by right-clicking your project and selecting Properties. In the Properties Window, expand WebSphere Portlet Factory and select Project Deployment Configurations.

After you have created your project, and can always go back and modify your project properties. For example, you might decide that auto-deploy of the portlet WAR file is too slow, and you want to disable this feature. All of this is easily accomplished by modifying your project's deployment configuration.

Hint 3. Reduce the size of your WAR file by removing unnecessary files from being built into the WAR file.

Suppose you have several samples, test models, images, and HTML files that you used during development, but are not required for your production portlets to work. In order to reduce the size of your WAR file, you can exclude these files from being deployed. There are two ways to do this:

- Create a directory called `**/nodeploy/**`. Artifacts placed within this directory are not included in a deployed WAR file. Note that you can create one or more `/nodeploy/` directories at any location within a project. You can then place within these directories all of the project artifacts that you do not want included in the deployed WAR. When the WAR is built by the Ant scripts, the contents of the `/nodeploy/` directories will be excluded from the build process.
- Use the `.excludeFromServer` file. Artifacts listed in this file are not included in a deployed WAR file. There are two versions of this file:

- One is a master version located in your feature set directory - *`install_directory`*`\FeatureSets\Web-App_X.XX.X\Tools\antScripts\public\`. If you want to always exclude the same files from every project you create, then you should modify this master `excludeFromServer` file.
- The other version of this file is located at the root of every project you create. Since the most common scenario is to exclude artifacts on a project-by-project basis, you will tend to want to utilize the project-level `excludeFromServer` file.

Resources

This section provides other sources of information to assist you with WebSphere Portlet Factory.

- Builder help

  When you have a builder opened for editing, you can get detailed information about the builder and the inputs it provide using either the Help button at the bottom of the editor or the F1 key.

- IBM WebSphere Portlet Factory wiki

  Provides numerous articles and samples to demonstrate common techniques for developing models in WebSphere Portlet Factory, available at http://www-10.lotus.com/ldd/pfwiki.nsf

- IBM WebSphere Portlet Factory customer discussion forums

  http://www-1.ibm.com/support/docview.wss?rs=3044&uid=swg27011853

- IBM WebSphere Portlet Factory Red Book

  http://www.redbooks.ibm.com/abstracts/SG247525.html?Open

- Extension packages

  You can access additional documentation for the feature sets provided to support Domino, SAP, PeopleSoft, and Siebel back-end data sources from the Windows Start menu: **Programs ➔IBM WebSphere ➔Portlet Factory ➔Documentation**.

- IBM WebSphere Portal Information Center

  This set of documentation includes information about WebSphere Portal server administration, as well as portlet deployment and management. You can access all published versions of the documentation at http://www.ibm.com/developerworks/websphere/zones/portal/proddoc.html

- IBM developerWorks has more articles and resources on Portlet Factory. See the WebSphere Portlet Factory Zone on developerWorks for links to these resources: http://www.ibm.com/developerworks/websphere/zones/portal/portletfactory/

Trademarks

- DB2, IBM, Lotus, Tivoli, Rational, and WebSphere are trademarks or registered trademarks of IBM Corporation in the United States, other countries, or both.
- Windows and Windows NT are registered trademarks of Microsoft Corporation in the United States, other countries, or both.
- Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.
- Other company, product, and service names may be trademarks or service marks of others.

IBM copyright and trademark information: http://www.ibm.com/legal/copytrade.phtml