Get-Resources 4.1
Tailoring Kit Guide

For Windows

Peregrine
SYSTEMS®

Contents

Introducing the Get-Resources Tailoring Kit

The Get-Resources Tailoring Kit includes:

- Peregrine Studio
- Source files for Get-Resources

The Get-Resources is intended for Web application developers who are familiar with Extensible Markup Language (XML), ECMAScript, Structured Query Language (SQL), and back-end database systems such as AssetCenter and ServiceCenter.

Peregrine Studio is a graphical development tool that you can use to customize Get-Resources. Get-Resources consists of a series of Web-based interfaces that allow users to, for example, order and purchase goods, search for requests, and submit purchase orders. The Peregrine Portal common interface determines what portions of Get-Resources the user sees.

The Web-based interfaces are the result of the following components:

- A collection of XML form defintions that provide the browser interfaces for Get-Resources. The Get-Resources XML form definitions are created with Peregrine Studio and then dynamically converted into HTML at runtime.
- A Web server to host the Get-Resources JSP content.

- A Java-enabled application server to run the Archway servlet and convert XML form defintions into HTML. The Archway servlet routes and formats data requests between Get-Resources and the back-end database.
- A collection of ECMAScripts that allow for dynamic parsing and formatting of Get-Resources data sent to and received from the client Web browser.

The Get-Resources files produced during a build are the result of the following Peregrine Studio components:

- A project file that describes Get-Resources. Each project file contains only the code necessary to produce and deploy Get-Resources.
- A collection of XML form defintions that define the functionality of Get-Resources. The Get-Resources XML form defintions are built in Peregrine Studio and deployed to the application server at runtime.
- A back-end database or application to store the data accessed by Get-Resources forms, track workflow tasks, and store personalization changes.
- Document schema definitions used to format message objects between the Archway servlet and the back-end database. All message objects are formatted as XML documents.
- ECMAScripts to generate and send message objects to the Archway servlet. The messenger objects can be used to query the back-end database for specific data and format the results for display in Get-Resources forms.

About this Guide

This guide is intended for use by a developer who will be tailoring Get-Resources from the source code provided with the tailoring kit.

This guide should be used in conjunction with several other manuals, which are:

- The Get-Resources installation, administration, and basic tailoring guides.
- The back-end database documentation for your installation.
- The application server documentation for your installation.

Conventions Used in this Guide

Screen shots in this guide are included as examples only. Get-Resources forms are shown using the Classic theme.

The following documentation conventions are used in this guide:

Object	Example
Button	Click **Next**
File name	The login.jsp file

Sample script or XML code	```var msgTicket = new Message("Problem");```
	```...```
	```msgTicket.set("_event", "epmc");```
	
	The ellipsis (...) is used to indicate that portions of a script have been omitted because they are not needed for the current topic. Samples of code are not entire files, but they are representative of the information being discussed in a particular section.

Menu option	Select Start>Program Files.
Book title	Refer to the *Get-Resources Installation Guide*.

SECTION

Setting up a Development Environment

This section describes how to install and use the Get-Resources Tailoring Kit development environment.

This sections includes:

1 Installing the Get-ResourcesTailoring Kit

CHAPTER

The Get-Resources Tailoring Kit installation allows you to install a JDK, Peregrine Studio, and the source files for Get-Resources.

Before you begin the installation, you should have already installed Get-Resources and any application servers and back-end systems required.

This chapter covers the following topics:

- *Installing the Get-Resources Tailoring Kit* on page 16
- *Opening the Get-Resources project* on page 19
- *Setting up a tailoring environment* on page 19

Installing the Get-Resources Tailoring Kit

The following sections describe how to install the Get-Resources Tailoring Kit on a Windows system.

Note: The Get-Resources Tailoring Kit does not run on UNIX, although files built by the Tailoring Kit can be deployed to a UNIX system.

Tip: Do not install the Get-Resources Tailoring Kit on your production system. Instead, install the tailoring kit on a development environment and then deploy your changes to your production environment after you have had a chance to test them.

To install the Get-Resources Tailoring Kit:

1 Insert the Get-Resources installation CD into the CD-ROM drive.

The Get-Resources Tailoring Kit splash screen opens displaying a list of installation options.
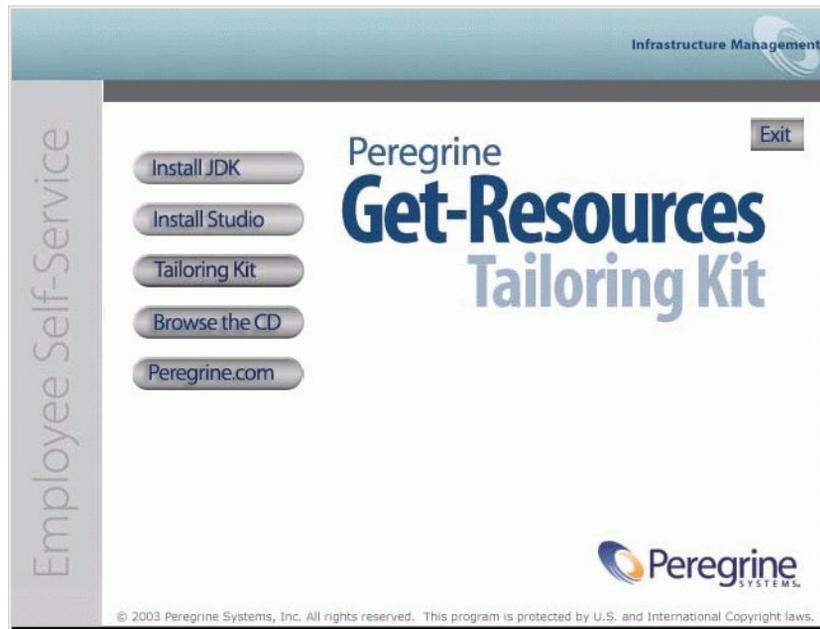
2 Install the required platform components for the Get-Resources Tailoring Kit.

- ■ **Install JDK.** Click this button to install the Java 2 SDK 1.3.1_05 on your system.
- ■ **Install Studio.** Click this button to install Peregrine Studio version 2.2.0.1068 on your system.

3 Click **Tailoring Kit**.

The Get-Resources Tailoring Kit installer opens.
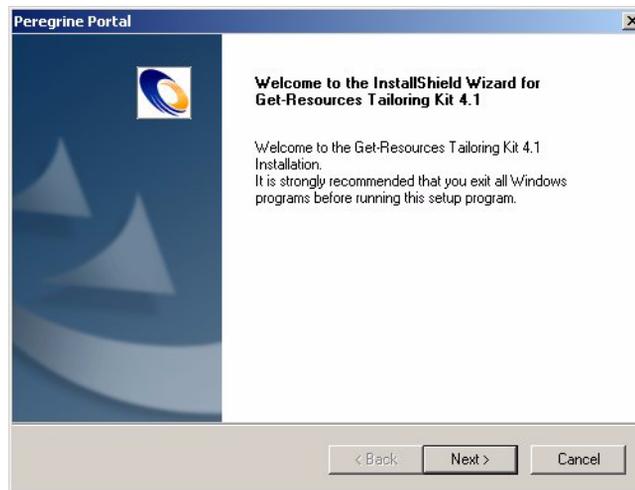
4 Click Next to continue.

The Choose Destination Location page opens.
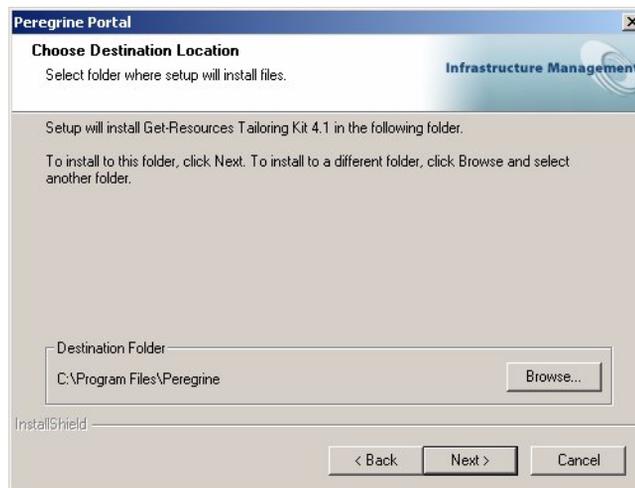
5 Click **Next** to accept the default installation location, or click **Browse** to select another installation location, and then click **Next** to continue.

The installer copies and deploys the files to your system and then the InstallShield Wizard Complete page opens.
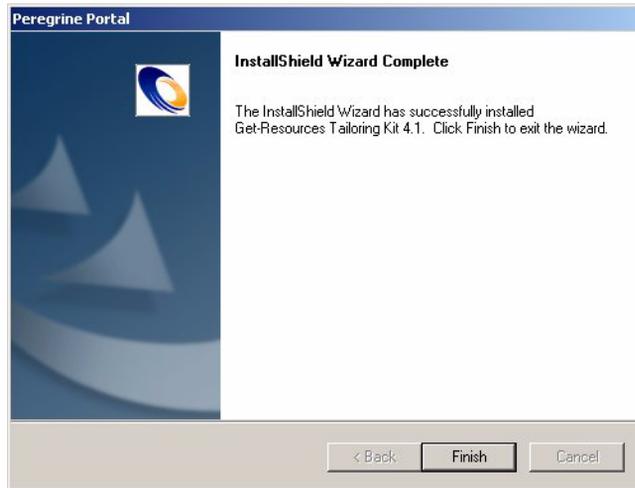
6 Click **Finish** to close the InstallShield Wizard.

Opening the Get-Resources project

After the installation is complete, you can open the Get-Resources project in Peregrine Studio using the following procedure.

Important: If you have not already received a Peregrine Studio authorization file, contact Peregrine Customer Support. You will need this file in order to edit your Get-Resources files.

To open the Get-Resources project in Peregrine Studio:

1 Click **Start** > **Programs** > **Peregrine** > **Studio** > **Peregrine Studio**.

   Peregrine Studio opens.

2 Click **Tools** > **Authorization file**.

3 In any text editor, open the authorization file provided for Peregrine Studio.

4 Copy the contents of the authorization file into the Authorization file dialog box in Peregrine Studio.

5 Click **OK**.

6 Click **File** > **Open project**.

7 Browse to the location of your Get-Resources project file (.adw file). For example:

   C:\Program Files\Peregrine\Get-It Tailoring Kit\get-resources

8 Select your Get-Resources project file:

   ■ Get-Resources.adw

9 Click **Open**.

Setting up a tailoring environment

You can set up one or more development environments separately from your production environment. A development environment lets you modify and build Get-Resources on a separate computer system than your test or production environments.

Setting up a development environment

You need the following minimum components for a Get-Resources Tailoring Kit development environment:

- Peregrine Studio.
- Java Runtime Environment 1.3 or later (necessary to run Studio), or the Java Development Kit provided with your Web application installation.
- Get-Resources Tailoring Kit (includes the Get-Resources source files).
- Java SDK 1.2.2 or later if you want to create or edit your own wizards for Peregrine Studio.

With this minimal development environment, you can modify Get-Resources using the built-in Peregrine Studio tools and wizards. You can then do one of the following:

- Build your Get-Resources projects on the development computer and copy the results to a production environment.

or

- Enter the network path to the production environment in your Peregrine Studio Build Settings.

Important: If you are using source control software to store your project files, you will need to configure your Peregrine Studio to check out and check in the source files. You can add your source control settings from **Tools** > **Options** > **Source control**.

Setting up a testing environment

You need the following components to test or debug your modifications:

- Peregrine Studio.
- Java Runtime Environment 1.3 or later (necessary to run Peregrine Studio).
- Get-Resources Tailoring Kit (includes the Get-Resources source files).
- If you want to create or edit your own wizards for Peregrine Studio, you will need to install a Java SDK 1.2.2 or later. The Java 2 SDK Standard Edition v1.3.1_05 is provided on the Get-Resources Tailoring Kit installation CD.
- An installed instance of Get-Resources including the following software:

- A Web server. Apache is provided on the Get-Resources installation CD.

- A Java-enabled application server. Tomcat is provided on the Get-Resources installation CD.

- JavaScript-enabled Web browser (necessary to view changes to Get-Resources). See the latest compability matix on the Peregrine support site for a list of supported Web browsers.

With this testing environment, you can build and view your changes from a single computer. To set up a testing environment, you must install both Get-Resources and the Get-Resources Tailoring Kit. Refer to the *Get-Resources Installation Guide* for instructions and requirements for installing Get-Resources.

Tip: You can save multiple versions of Get-Resources in separate project files. When you are ready to test a particular tailored version, you can load the tailored project, build it, and deploy it to your test environment.

2 Using Peregrine Studio

CHAPTER

This chapter provides an overview of the Peregrine Studio interface. For more information about configuring or using Peregrine Studio, refer to the Peregrine Studio online help.

This chapter covers the following topics:

- *The Peregrine Studio interface* on page 24
- *Best Practices* on page 29

The Peregrine Studio interface

The Peregrine Studio interface includes:

- Project Explorer
- Properties window
- Edit toolbar
- General information display
- Contextual help
- Address
- Package selector
- Advanced information

All elements of the interface except the Project Explorer and the Properties Window can be hidden by clearing them on the View menu.
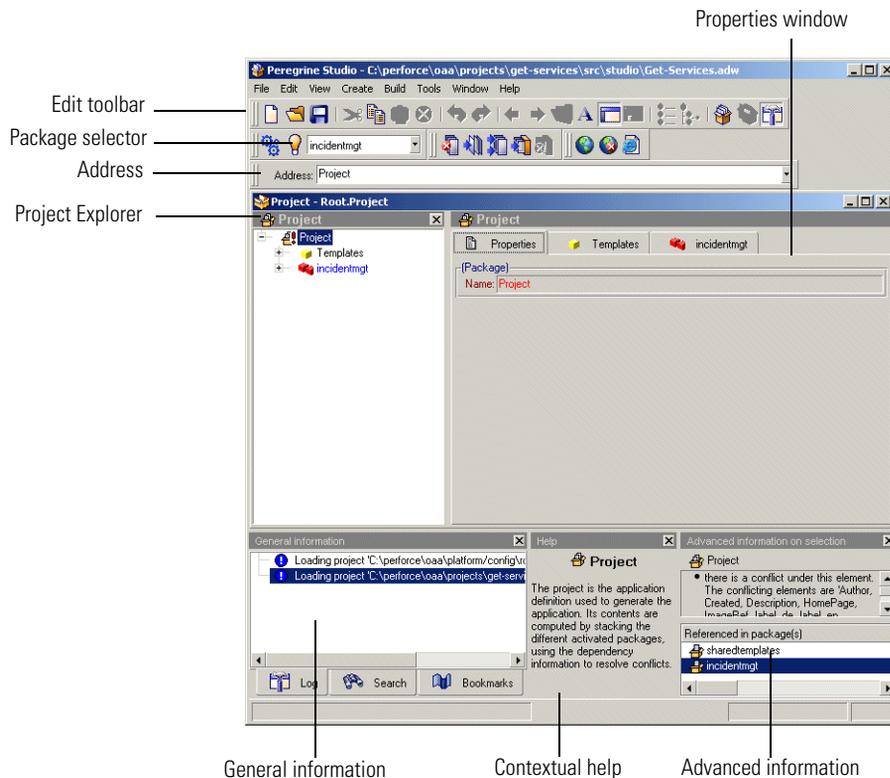
Project Explorer

The Project Explorer provides a hierarchal view of all the components that comprise a Peregrine Studio project. The Project Explorer window displays each component as a separate node within the tree.
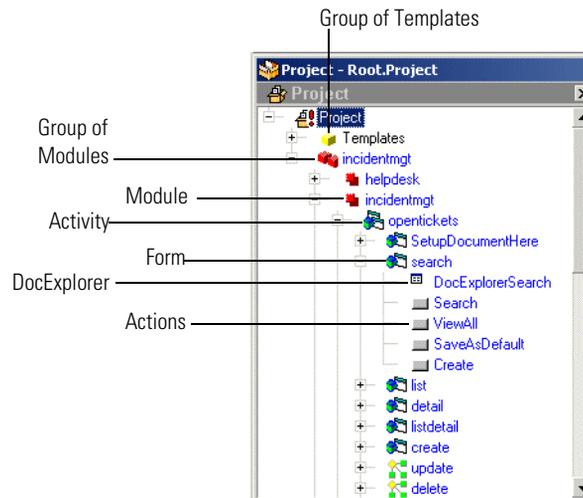
Left-click a node

Click the node listing the component you want to change and the properties of the component display in a window of the Properties pane.

Right-click a node

Right-click a node to display a list of context-sensitive options.

The options listed in the following table are available for all nodes.

Menu item	Description
New	Provides a context-sensitive menu of allowed components that you can add from the current node. The list of components in this menu is dynamically updated for each node of the Project Explorer tree.
Open	Displays the properties of the selected component in a window of the Properties pane.
Open in New Window	Displays the properties of the selected component in a new window of the Properties pane.
Rename	Renames the selected node to the new name typed by the user. This option will only be available when a package extension has been activated as the save location for changes.
Cut	Removes the selected node, and all child nodes underneath, and places a copy in the Windows clipboard.
Copy	Copies the selected node, and all child nodes underneath, to the Windows clipboard.
Paste	Inserts the contents of the Windows clipboard. If the clipboard contains a Studio component, it will be automatically placed within the tree according to the type of component it is.
Delete	Deletes the selected node and all child nodes. This option will only be available when a package extension has been activated as the save location for changes.
Help	Displays the Studio help system.
Export node	Saves a copy of the selected node, and all child nodes underneath, as an XML file, which can be imported into a Studio project.
Import node	Opens a user-selected XML file describing Studio nodes and inserts it into the tree. The imported node will be inserted below the node you right-clicked.
Add Bookmark	Adds a bookmark link to the node you currently have open in Studio. If you browse to another location and then want to return to this node, click the Bookmarks tab in the General Information window and select the appropriate bookmark.

The following image shows how some of the common Peregrine Studio components are displayed in a Peregrine Web application interface.
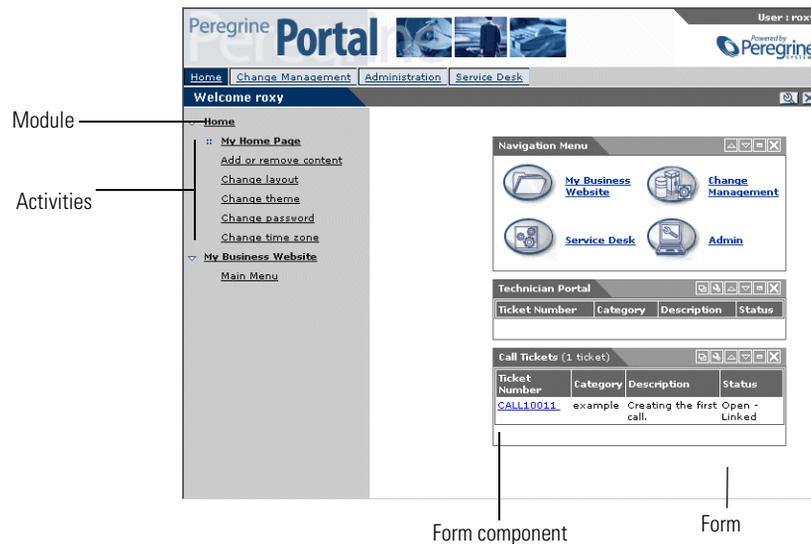
The address bar

You can use the Address Bar to navigate directly to any Peregrine Studio project component. The address bar will display as a text box below the Edit Toolbar.

To display the address bar:

1 Open Peregrine Studio.

2 Click View > Address.

The Address Bar displays below the menus.

Drag and drop

Peregrine Studio supports drag and drop movement of components within the Project Explorer. Changing the order of nodes in the Project Explorer will change how the items are presented in the Peregrine Studio build.

To move a component within the Project Explorer:

1 Click and hold the left mouse button over the name of the node you want to move.

2 Drag the node to the new location in the Project Explorer tree.

The node appears underneath the component (of the same level) where you drop the node.

Note: You cannot move components out of the order enforced by the DSD. For example, you cannot move a form out of an activity and place it at the same level as a module. You can, however, change the order of the forms listed under an activity.

Best Practices

The following recommendations will make tailoring projects easier and reduce the amount of troubleshooting you need to do.

Avoid changing form definitions outside of Peregrine Studio

Although Get-Resources form definitions are XML files, the XML grammar used to build them is specific to Peregrine Studio. If you make changes to the Get-Resources form definitions outside of Peregrine Studio you risk corrupting your project file and complicating your troubleshooting efforts. If you want to view the XML form definitions, you can safely enable the source view from within Peregrine Studio.

The source view does not support direct editing of the XML form definitions. All XML source views are listed with a grey background which indicates that the item is read-only.

To view the XML source code within Peregrine Studio:

1   Select the node of the Web application or component you want to view from the Project Explorer.

2   Click the **Source view** button **A** (the blue capital A).

The XML source appears in the Properties window. The XML source code is color coded as you define in the project settings.

Avoid enabling advanced options

The advanced options found in Tools > Options > Advanced change the way your project is protected and built. In general, Peregrine recommends that you avoid enabling all advanced options except the HTTP Listener. Enabling any other options may overwrite needed source files in your Get-Resources project and complicate your troubleshooting efforts. Furthermore, Peregrine cannot support any changes you make to the source packages delivered with Get-Resources.

Avoid using the clean the target folders build option

The **Clean the target folders** build option deletes all files in your build folder. If you build directly into your application server's deployment folder, using this option will delete the files necessary to run Get-Resources and require you to reinstall Get-Resources. You should only consider using this option if you install the Get-Resources Tailoring Kit on a different machine than your Get-Resources installation.

Clear your application server cache every time you build changes

To ensure that you always see the latest changes in your test environment, Peregrine recommends that you clear your application server's cache. This is especially important if you use Tomcat 4.1.x as your application server.

Create new or change existing templates to apply global changes

Your Get-Resources project contains a Group of Templates node where you can store and change preconfigured form components. Each form that uses a template inherits the properties of the template. If you want to make global changes to Get-Resources, search for the relevant templates in the Group of Templates node. If you want to create a re-usable collection of form components you can create a new template to store your changes. Any template you create appears as an option in the New context-sensitive menu.

To add a template to a form:

1 Click on the node that you want to add a template component.

2 Click **Create** and then select the template name from the list beneath the horizontal rule. You can also Right-click on the node and select **New**.

The Create and New menus display only the templates that are valid for the location you selected.

Important: Do not drag and drop or copy and paste a template into a form. In order for Peregrine Studio to recognize the template you must add the template form components from the New menu.

Enable the HTTP listener and display form information options

Using the HTTP Listener, you can click on the Form Information address listed for a given form and the appropriate form properties will be displayed in Peregrine Studio. This debugging feature allows you to navigate through Get-Resources with a browser and quickly bring up any particular form that needs modification.

Important: The HTTP Listener cannot bring up the administration, home page forms, nor any Get-Resources forms that are built using DocExplorer. The source code for these three modules is no longer provided with the Get-Resources tailoring kit. You can tailor such forms directly using Personalization.

To enable the HTTP Listener:

1 Open Peregrine Studio.

2 Click Tools > Options > Advanced.

3 Select the **Use listener** check box from the HTTP Listener section.

4 Select the port number you want the HTTP listener to use (the default port is 81), and then click **OK**.

5 Save your Peregrine Studio project.

6 Close and then re-open Peregrine Studio to initialize the HTTP listener.

7 Open the project file containing the form you want to change in Peregrine Studio.

   Note: Be sure to select or create a package extension in which to save any changes.

To enable the Form Information functionality:

1 Log in to Get-Resources as an administrator, or access the Admin module directly from the Administrator login page (`admin.jsp`).

2 Click **Admin** > **Settings** to display the Settings form.

3 On the **Logging** tab, set the **Show form info** setting to *true*.

4 Click **Save** at the bottom of the form to activate your new settings.

5   On the Control Panel form of the Admin module, click **Reset Server** to commit your changes.

Click this link to open the form in Studio.

6   Navigate to the form you want to tailor.

7   Click the Peregrine Studio address displayed in the Form Information banner of the Web application form.

Peregrine Studio will appear as the active window and display the current form's properties page.

Set the color for your extension changes

By default, all changes or additions you make to your Peregrine Studio project are highlighted with blue text. You can change the color Peregrine Studio uses to indicate extension changes with the following procedures.

To change the color Peregrine Studio uses to indicate extension changes:

1   Click **Tools** > **Options** > **Appearance**.

The appearance window opens.

2   From the **Extension color** drop-down box, select the color you want to use to indicate changes to base packages originating from package extensions.

3   Click **OK**.

Within the Project Explorer view, Peregrine Studio highlights each node of the tree that contains a component that has been changed or added. This allows you to navigate through the Project Explorer tree view and locate where you have made changes and additions.

To view the changes made in a project:

1 Select a node displayed with blue text to view the component properties.

2 Review the properties listed in the window displayed to the right of the Project Explorer (the Properties window). Changes that were made to this component will be displayed with blue text. If no blue text is displayed in the Properties window, then the change or addition is in one of the child nodes below the current node.

3 If necessary, expand any child nodes highlighted with blue text and review the Properties window for changes.

View referenced components with the lookup button

Whenever an item links to or references another component, Peregrine Studio displays a lookups button ![lookups button] next to the field.

You can click this button to display the form, image, schema, or script that is called by the reference.

Use Go to Previous View ![orange arrow] (the orange arrow pointing to the left) to return to the component making the reference.

3 Peregrine Studio Projects and Packages

CHAPTER

Peregrine Studio *projects* contain all of the *packages* that make up an application. A new package must be created when you are making changes to your project. You can then activate or deactivate packages depending on the features you want to be included in your current project.

This chapter includes the following topics:

Peregrine Studio projects

Peregrine Studio saves all the source files for Get-Resources as a project file. A Studio project file consists of the following components.

Studio component	Description
Get-Resources components	The XML form definitions that specify the functionality of the Get-Resources interface. The application server will dynamically render the Get-Resources XML form definitions as HTML when a specific form is requested.
ECMAScripts*	ECMAScripts create and format message objects to the Archway servlet. Get-Resources components will use ECMA message objects to display and process data.
Document schema definitions	The XML files that define how the Archway servlet should format the ECMA message objects sent to and received from back-end databases. Get-Resources components will use the ECMA message objects to display and process data.
Presentation files	Any supporting files such as images, client-side JavaScript, hand-coded HTML or JSP files, or translation strings that will be included with Get-Resources.
Stylesheets	The Cascading Style Sheet (CSS) files that define the colors and fonts that will be used in your Get-Resources pages.

* ECMAScript is the core language standard shared between the JavaScript and JScript libraries.

For a listing of where Peregrine Studio saves and builds these files, see *Project files* on page 40.

Project components

Peregrine Studio organizes project components into a hierarchy of parent and child elements. The position of a project component determines the individual properties it can have. Properties include, for example, what other project components can be placed within the component and the type of editor used to edit the component. All Peregrine Studio projects conform to the hierarchy listed below:

Template

Group of modules

    Module

        Activity

            Form
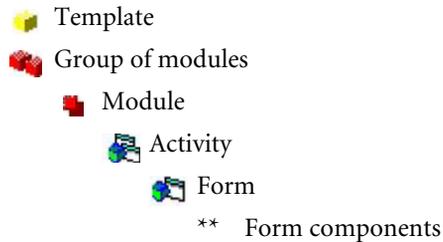
                **   Form components

Project component descriptions

This table lists and describes some of the common Peregrine Studio components. For a complete list of the components that make up a Peregrine Studio project, see *Peregrine Studio Components* on page 53.

Component	Description
Project	The project component: ■ is the container for all the elements that are part of your current project file. ■ is always the top node of the Project Explorer tree. ■ is represented by an open package icon ( ) in the Peregrine Studio Project Explorer tree.
Templates (support files)	The templates component: ■ is the container for all the common elements reused throughout the project. ■ appears with a yellow cube icon ( ) in the Peregrine Studio Project Explorer tree.

Component	Description
Group of modules	The group of modules component: ■ is the container for all the XML form defintion files and modules that make up Get-Resources. ■ appears with a double red cubes icon (　) in the Peregrine Studio Project Explorer tree. ■ does not have any one dedicated graphical representation in the built project.
Module	The module component: ■ is a container for the activities and forms that make up Get-Resources. ■ appears with a double red box icon (　) in the Peregrine Studio Project Explorer tree. ■ appears as a text link on the navigation sidebar and may also appear on the Get-Resources Home Menu. **Note:** The module component is usually where access restrictions are defined. Setting access restrictions limits a module to particular user roles.
Activity	The activity component: ■ defines a particular task or action such as searching for records, displaying records, or entering records. ■ is a container for a particular set of forms. ■ appears with a cube and two window panes icon (　) in the Peregrine Studio Project Explorer tree. ■ appears as a text link on the navigation sidebar (Activity Menu).
Form	The form component: ■ is where Get-Resources user interfaces and displays are defined. ■ appears with a cube and a single window pane icon (　) in the Peregrine Studio Project Explorer tree. **Note:** Typically, the system displays each form component as a page in the main frame.

Component	Description
Form components	Form components such as fields, actions, tables, and lookups: ■ define the actual user interfaces and displays used in a Get-Resources form. ■ appear with a variety of icons in the Peregrine Studio Project Explorer Tree. ■ typically have a graphical element in a Get-Resources form.
Group of scripts	The group of scripts component: ■ is a container for all the server-side ECMAScripts used by Get-Resources. ■ appears with a document with a yellow border icon ( ) in the Peregrine Studio Project Explorer tree.
Group of schemas	The group of schemas component: ■ is a container for all the document schema definitions that Get-Resources uses. ■ appears with a data store and document icon ( ) in the Peregrine Studio Project Explorer tree.
Group of files	The group of files component: ■ is a container for supplemental files that your Web applications can use. You can store images, client-side JavaScript, localized string files, or initialization files here. ■ appears with a folder icon ( ) in the Peregrine Studio Project Explorer tree.
Group of Strings	The group of strings component: ■ is a container for all the text strings that Get-Resources uses. ■ appears with a globe icon ( ) in the Peregrine Studio Project Explorer tree.

* Portal Components are available only in the portal module.

Project files

This table describes the files that make up a Studio project and the information they contain. Items listed in *italics* are variables. To determine the actual file name, replace the italic text with the component name.

Warning: Do not edit these files outside of Studio. Manual changes you make outside of Studio will be lost during the build process.

Component	Save and build location	Contains
project	■ Saved as: C:\Program Files\Peregrine\ Get-It Tailoring Kit\get-resources\ *project*.adw	■ \<package\> names ■ Path to *package*.xml
package	■ Saved as: C:\Program Files\Peregrine\ Get-It Tailoring Kit\get-resources\ *package**package*.xml	■ \<package\> name ■ \<modules\> name ■ \<module\> names ■ Path to *module*.xml ■ Schema Names ■ Path to *schema*.xml ■ Script Names ■ Path to *script*.xml ■ String Resources
modules	■ Saved as: C:\Program Files\Peregrine\ Get-It Tailoring Kit\get-resources\ *package**package*.xml	\<modules\> name
module	■ Saved as a single file: C:\Program Files\Peregrine\ Get-It Tailoring Kit\get-resources\ *package*\modules*module*.xml ■ Built as a collection of forms: C:\OAA\build\WEB-INF\apps\ *package*\forms*module**activity*\ *form*.xml	■ \<module\> name ■ XML code for \<activity\>, \<form\>, and \<form\> components

Component	Save and build location	Contains
schema	■ Saved as C:\Program Files\Peregrine\ Get-It Tailoring Kit\get-resources\ *package**modules*\Schemas\ *schema*.xml ■ Built as: C:\OAA\build\WEB-INF\apps\ *package*\schema*schema*.xml	XML code for <schema>
script	■ Saved as: C:\Program Files\Peregrine\ Get-It Tailoring Kit\get-resources\ *package**modules*\ Scripts*script*.xml ■ Built as: C:\OAA\build\WEB-INF\apps\ *package*\jscript*script type**script*.js	XML code for <script>
presentation files	■ Saved as: C:\Program Files\Peregrine\ Get-It Tailoring Kit\get-resources\ *package**modules*\presentation\ *presentation file*.jsp ■ Built as: C:\OAA\build*presentation file*.jsp	Directory where presentation files can be stored to be included in a Studio build.
strings	■ Saved as: C:\Program Files\Peregrine\ Get-It Tailoring Kit\get-resources\ *package**package*.xml ■ Built as: C:\OAA\build\WEB-INF\apps\ *package**package*_en.str	Text strings for English, German, Spanish, Italian, and French.

Building a project

During the build process, Studio compiles all project files and copies them to deployment folder you specified in your Peregrine Studio Build Settings.

Build options

Peregrine Studio offers the following build options from the **Build** menu:

Build option	Description
Clean the target folders	Deletes the contents of the presentation and deployment folders.
Build element	Builds the currently selected element in the project explorer. This element will not be rebuilt the next time a differential build is performed.
Differential build	Builds only those elements that have changed since the last build.
Rebuild all	Builds all elements of the project.
Stop Build	Stops a currently running build process.

Warning: The **Clean the target folders** option cleans the folders listed in your build settings. If you build your project directly to your application server, then using this option deletes your installation of Get-Resources. It is recommended that you avoid using this option if you have installed Get-Resources on your development machine.

Setting project build settings

You can define the build settings option to define the file locations and file formats used during the build process. Each Peregrine Studio project can have its own project settings.

To set project build settings:

1 From the Build menu, select **Project settings**.

The Project Settings window opens.
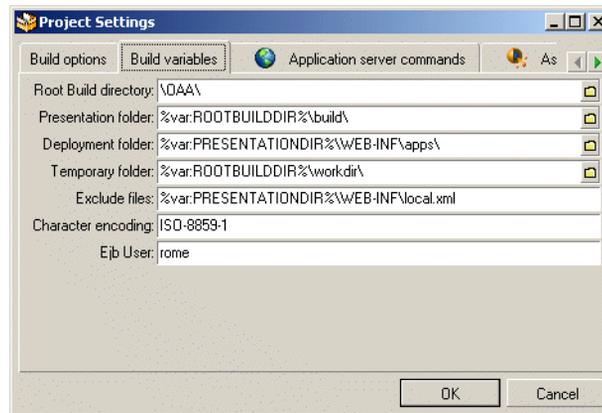
2 Click the **Build Variables** tab.

3 Enter or browse to the proper directory for the following settings.

   a Root Build Directory—This is the drive and folder you want to be root for building Peregrine Studio projects. Whatever path you enter here becomes the variable %var:ROOTBUILDDIR%.

   b Presentation folder—This is the folder where your application server will look for files to serve. If you are running an application server on the build machine enter the full path to your application server context root here. For example:
   c:\Program Files\Peregrine\Common\Tomcat4\webapps\OAA
   If you do not have an application server on the build machine, you may enter any path where you want files deployed. Whatever path you enter here becomes the variable %var:PRESENTATIONDIR%.

   c Deployment folder—the folder where scripts, schemas, and XML form definitions are located. You do not need to change the value of this setting.

Warning: Do not change this option.

   d Temporary folder—the folder where Peregrine Studio will generate temporary files used in the build process. You do not need to change the value of this setting.

e Exclude files—a semicolon-separated list of files or directories that you want Peregrine Studio to exclude from removing or rebuilding during a build. You do not need to change the value of this setting.

f Character encoding—Not used. JSP encoding is determined by the character encoding setting on the Settings page of the Admin module. You do not need to change the value of this setting.

g Ejb User—Not used. Get-Resources does not use the rome adapter. You do not need to change the value of this setting.

4 Click **OK** to save your settings.

Peregrine Studio project packages

Packages contain all the XML form defintions, ECMAScripts, and schemas necessary to run Get-Resources. Your Get-Resources project is defined by one or more packages, which are either *system* or *extension* packages:

- System packages. The system packages provided by Peregrine define the out of the box functionality of Get-Resources.

- Extension packages. Any packages you create are called *extensions*. Package extensions store all of your additions or modifications to the existing system packages.

You can see the system packages and the extensions that make up your project from the Package Activation toolbar. This view displays the active packages that can be edited and built in your project. When a package is activated, the changes or additions will be included in the build. When a package is deactivated, the changes or additions will not be included in the build. The modular design of packages allows you to decide which changes and additions will be included or excluded from the build process.

Tip: Group similar Web application functions in the same package extension. This will allow you to activate or deactivate groups of functions using the Package Activation toolbar. For example, if you are testing different interfaces with the same functionality, you may want to save each interface in a different package extension. After you determine which interface is better, you can implement the new interface by activating that package extension and rebuilding the project.

Packages are not displayed in the Project Explorer *Project* tree. The list of available packages (packages that have been activated) is included in the Package Explorer drop-down list located below the toolbar in Peregrine Studio.

Saving changes with package extensions

All additions and changes to a project must be saved under a package extension name. By default, all of the system packages that ship with Get-Resources are write-protected and cannot be used as the save location for your tailoring efforts. To tailor your installation you need to create one or more new package extensions where your changes and additions will be saved.

To create a new package extension:

1 Open Peregrine Studio.

2 Click **File** > **New package** to start the Create New Package wizard.

3 Enter the name and package dependencies for the new package.

   a Name. Enter a name for the new package extension. The package extension name cannot contain spaces or special characters.

   b Dependencies. Select the existing system package or packages that your package extension will be dependent on. Select the system packages that you want to make changes to as the package dependencies. Your new package extension must be dependent on at least one existing package. See *Package dependencies* on page 47 for more information on package dependencies.

4 Click **OK** to complete the wizard.

5 Save your Peregrine Studio project file.

6 Close and then restart Peregrine Studio.

Any changes or additions you make to Get-Resources will now be saved in your new package.

Activating and deactivating packages

You can control the packages and package extensions that are part of Get-Resources by activating or deactivating them from the Package Activation menu. To include a package in Get-Resources installation, activate the package, and then build the Studio project. To remove a custom package from your installation, deactivate the package and delete it from the following path:

C:\Program Files\Peregrine\Get-It Tailoring Kit\get-resources\package name.

Tip: In some cases it is simpler to re-install Get-Resources than to delete unwanted custom packages.

To activate a package:

1 To display the package activation toolbar, click **View**, and then click **Package Selector**.

The Package Activation toolbar is displayed.

2 Click the **Package activation** button ( ).

3 Select the checkbox next to the package name or names you want to activate.

4 Click **OK**.

All active packages will be included in the next build.

To deactivate a package:

1 Click the **Package activation** button ( ).

2 Clear the check box next to the package name or names you want to deactivate.

3 Click **OK**.

All deactivated packages will be excluded from the next build.

Important: Deactiviating a package does not delete it from the Get-Resources interface if you have already built it. To delete tailoring changes you have already built you can either re-install or delete the XML form definitions for your package extension.

Package dependencies

Each package has a list of dependencies that define what other packages it can make changes or additions to.

When you create a package extension, you must select the other packages that your extension can change. You will only be able to make changes or additions to the packages that are listed in your extension's package dependencies. If you try to make changes outside your extension's dependencies, you will produce a dependency conflict.

You can use the package dependency list to determine what other packages a particular extension affects. This information is particularly useful if you are trying to resolve conflicts in your projects.

Package dependencies are first defined by the New Package wizard when you create a package. You can manually change the package dependencies using the procedures described below.

Setting package dependencies

To set package dependencies:

1 Go to **Tools** > **Package Dependencies**.

2 From the left pane, select the package name for which you want to set dependencies.

   The list of defined dependencies appears in the right pane.

3 Select the check boxes next to the package names you want to add as package dependencies. Clear the check boxes next to the package names you want to remove as package dependencies.

   Note: Dependent packages activate or deactivate as a group. For example, suppose you create a user extension called *New_Interface* that is dependent on the *Extension* package. If you deactivate the *Extension* package, you will also deactivate the *New_Interface* package. If you activate the *New_Interface* package, you will also activate the *Extension* package.

4 Click **OK** to set the dependencies.

Warnings for conflicts

Peregrine Studio validates your project and ensures that there are no conflicting instructions or missing components. If Peregrine Studio encounters a conflict, it displays an exclamation point ❗ icon next to each node that contains a conflicting component within the Project Explorer view.

Peregrine Studio will display a conflict warning if any of the following conditions occur.

- Two or more active project components describe the same thing. For example, if you have two active package extensions that rename the same button, you will create a resource conflict.
- You make changes or additions to a package that is not defined as a dependent package. For example, if you create a package called *test* that is solely dependent on the package *changes*, then the *test* package cannot make changes or additions to other packages, such as *incidentmgt*. Attempting to make such changes will create a dependency conflict.

Resource conflicts

Resource conflicts occur when two or more activated package extensions describe the same project components. For example, if the Extension package extension adds a *submit* action to a form, then you will see a resource conflict if another package extension (for example, called *demo*) also adds a submit action to that form. The submit action on that form can only be described by one package extension at a time.

Resolving Resource Conflicts

To resolve a resource conflict, you can either deactivate the package extension with the conflicting project component or you can delete the project component creating the conflict from one of the package extensions. Continuing the example from above, you could either deactivate the *demo* package extension or you could delete the submit action from the *demo* package extension.

Dependency Conflicts

Dependency conflicts occur when you change a project component in a packages that is not listed as a dependency for your current package extension. For example, if the *demo* package extension is solely dependent on the *incidentmgt* package, then the *demo* package extension cannot make changes to the *sharedtemplates* package without creating a dependency conflict.

Resolving Dependency Conflicts

To resolve a dependency conflict you can either add a dependency to the package extension, or you can move the changes to another package extension with the proper dependencies. Continuing the example above, you could either make the *demo* package extension dependent on the *sharedtemplates* package or you could move the changes from the *demo* package extension to another package extension such as *extension*, which is already dependent on the *sharedtemplates* package.

Viewing Conflict Information

The Advanced Information pane tells you whether you have a resource or a dependency conflict.

To view conflict information:

1 Select a node with an exclamation point ❗ icon displayed next to the name from the Project Explorer view.

2 Click **View** > **Advanced information**.

A new information window will be displayed at the bottom of the Peregrine Studio interface. This window displays information on the conflict.

For additional information about a particular project component and its possible settings, refer to the *Studio Introduction* and the Studio online help.

Deploying tailoring changes

After you build your Peregrine Studio project file, you will need to deploy your new files to the application server running Get-Resources. The following sections describe how to deploy your tailoring changes to your test and production environments.

Deploying to Windows platforms

You can deploy your tailoring changes directly over your Windows network.

To deploy tailoring changes on Windows platforms:

1 Stop the application server on the target machine.

2 Copy the files from the Peregrine Studio deployment directory to the application server's deployment directory on the target server.

3 Restart the application server on the target machine.

Deploying to UNIX platforms

You can deploy your tailoring changes to UNIX platforms using whatever cross-platform methods you have available such as FTP, shared drives, or e-mail.
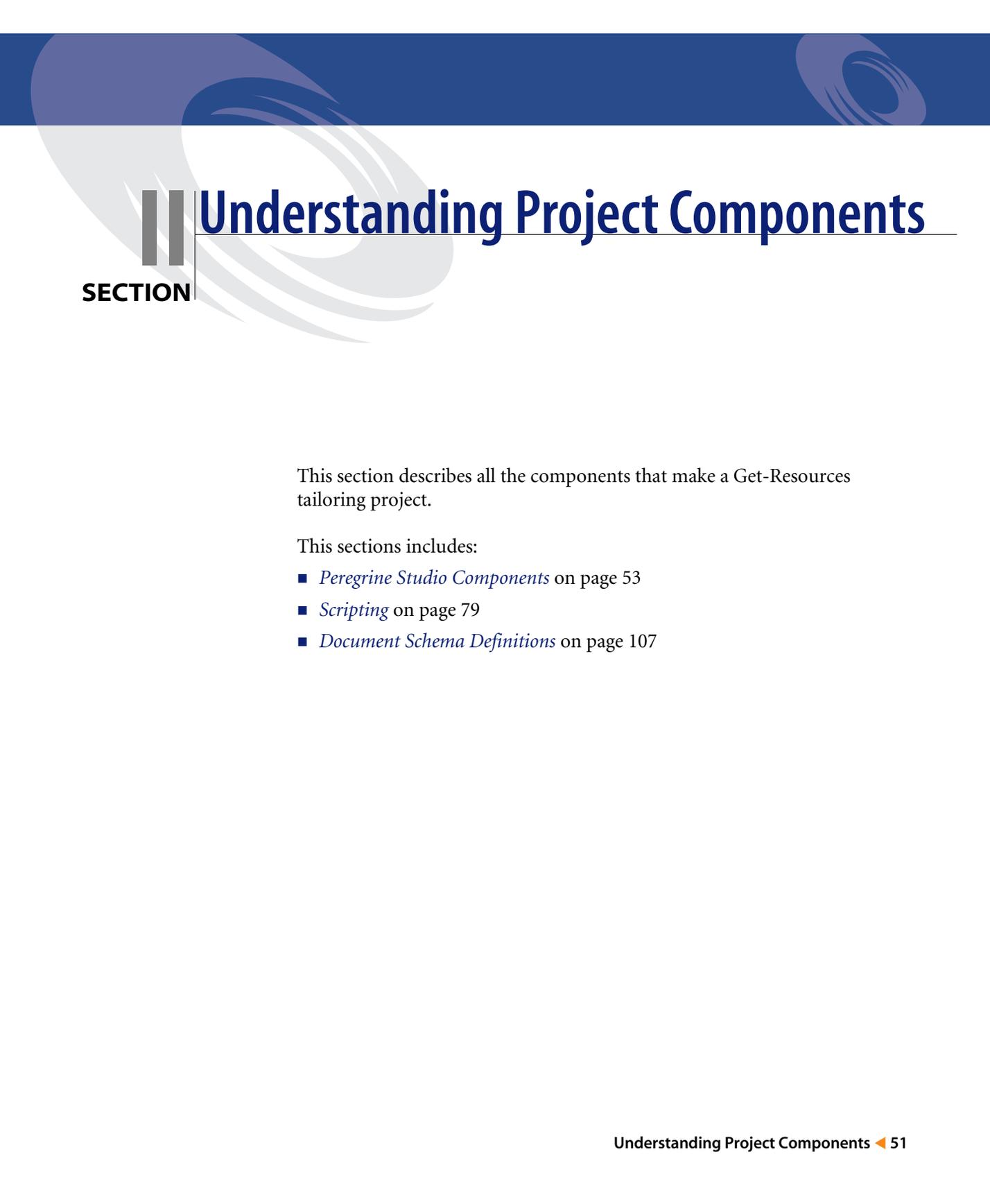
II SECTION

Understanding Project Components

This section describes all the components that make a Get-Resources tailoring project.

This sections includes:

- *Peregrine Studio Components* on page 53
- *Scripting* on page 79
- *Document Schema Definitions* on page 107

4 Peregrine Studio Components

This chapter contains a list and description of all of the components you can add to a Project in Studio. The information is grouped according to the menu structure with which these components are presented in Studio, following each component down to display all of the subcomponents available.

The menus displayed when you open the Get-Resources package in Peregrine Studio may vary slightly from the menu options documented here. Menu options change depending on the components you have created. For example, you must have the folder called shared templates in your package to enable DocExplorer Reference as a menu option.

This chapter covers the following topics:

- *Adding components* on page 54
- *Types of form components* on page 65

Adding components

To add components to your Project, right-click on the node to which you want to add a component, and a menu of options is displayed.
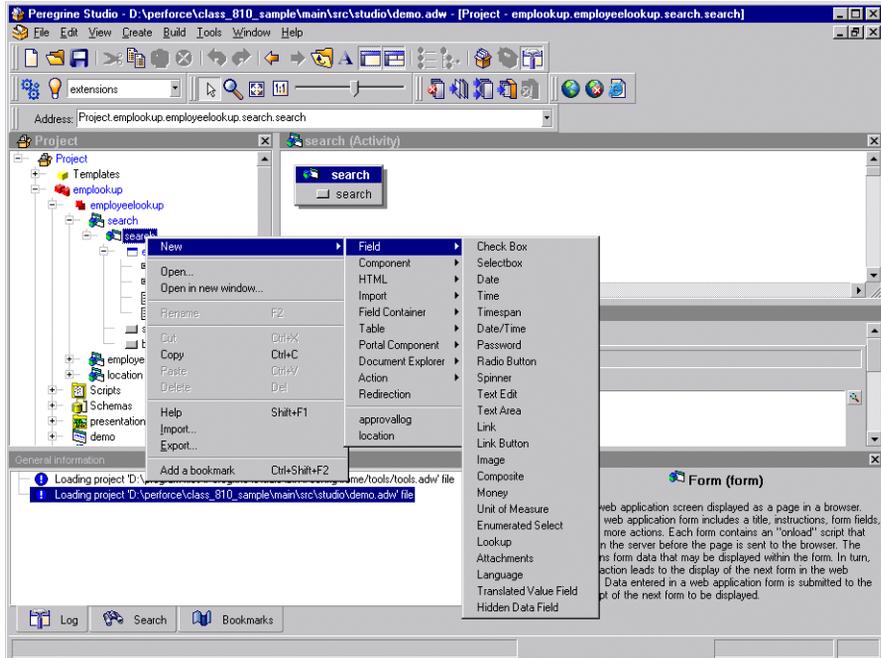
Project > New >

Directory Object—not supported.

Group of Modules > New >

When you create a Group of Modules component, it includes a folder called Explorers that contains default content for DocExplorer personalization screens. It also includes a Group of Roles, which is a list of roles that are used to control access rights. From the Group of Modules, you can create the following:

- **Module**—Get-Resources is organized into modules. Modules are often determined by the role that a user will take in performing tasks. For example, one module could be designed for employees who will be opening requests for service. Another module could be for managers approving requests. Modules are typically assigned specific access role restrictions so that only those users who need to perform the module's task have permission to do so.

  - **The Peregrine Portal** > **Activity**—Each module should contain one or more activities that define the steps users can take to complete the module's task. For example, a Request module could have activities for browsing catalogs, reviewing a shopping cart, and filling out a request form. Each activity is typically displayed in Get-Resources on a sidebar menu at the left of a form. Activities are typically assigned specific access role restrictions so that only those users who need to perform the activity's task have permission to do so.

    Form—Defines a Get-Resources screen displayed as a page in a browser. The typical form includes a title, instructions, form fields, and one or more actions. Each form contains an onload script that executes on the server side before the page is sent to the browser. The script obtains form data that may be displayed within the form. In turn, each form action leads to the display of the next form in Get-Resources. Data entered in a form is submitted to the onload script of the next form to be displayed.

    Field >

      Check Box—Allows the user to toggle a value on or off.

      Selectbox—Allows the user to select a value from a list displayed in a Combo Box field.

Date—Allows the user to view or enter a date. An optional calendar form component (Date Picker) can be enabled or disabled (the default is *enabled*) for users to enter dates. To define a start year for the drop-down list or for the calendar form component, add a + or - sign in front of a number. This number specifies the number of years before or after the current year you want the start and end years to be.

To designate a Date/Time calendar year start and end, add a + or - in front of a number to specify the number of years before or after the current year you want the start and end years to be.

Enable or disable the calendar form component.
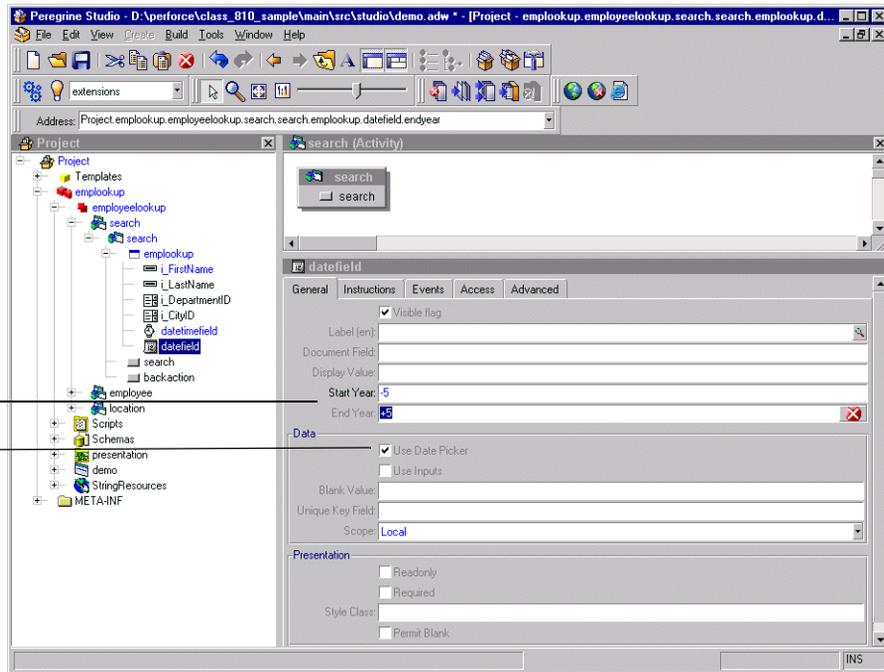
Time—Allows the user to view or set a time value.

Timespan—Allows the user to view or edit a timespan value.

Date/Time—Allows the user to view or set a date and time value. There is an optional calendar form component (Date Picker) that can be enabled or disabled in Studio (the default is *enabled*). See Date component.

Password—Allows the user to enter a password.

Radio Button—Allows the user to select one of several choices presented by radio buttons.

Spinner—Allows the user to enter a numerical value. The control allows the number to be typed in directly. It also allows the user to select a number by clicking on the spinner buttons that increase and decrease the value.

Text Edit—Allows the user to display or edit a value in a plain text field.

Text Area—Allows the user to enter text into a multiline edit field.

Link—Displays a hyperlink that the user can click on to navigate to another Web location or site.

Link Button—Displays an image button created out of background images and text.

Image—Displays an image.

Composite—Allows the creation of a field that consists of two or more fields placed next to each other.

Money—Allows the user to view or edit a monetary value.

Unit of Measure—Allows the user to view or edit a value that is a unit of measure.

Enumerated Select—Allows the user to select a value from a list displayed in a Combo Box field.

Lookup—Allows the user to enter a value by performing a lookup operation. The lookup is done in a separate pop-up window.

Attachments—Allows the user to view and add attachments to a document.

Language—Allows the user to select their preferred language from a list of supported languages.

Translated Value Field—Displays text returned by a translation script function.

Hidden Data Field—Stores data obtained by the form's onload script without displaying it to the user. The data is included when the form is submitted and the user navigates to another form.

Component >

Treelink—Displays a treelink component.

Directory—Displays a directory component based on data received from a document query to an adapter.

List Builder—Allows users to configure a list by selectively adding items to a listbox from a list of choices.

Workflow—Displays a workflow diagram.

OAA Workflow—Displays a workflow diagram.

Stack—Displays a stack component.

SVG—Displays an SVG component.

Web Application Menu—Displays a menu of all registered modules or packages in the current Web application.

HTML >

Blank Line—Adds a blank vertical line to the form.

Free-form HTML—Allows you to insert arbitrary HTML into a form. Can also be used to insert client-side JavaScript into a Web page, although large amounts of client-side JavaScript should be moved to a presentation file that can be imported by the page.

Import >

Static Import—Imports the text content of a file for inclusion in a Web page. For example, you can import files that define static HTML, JSP code or client-side JavaScript functions.

External HTML Plugin—Includes dynamic content into the form. At run time, the URL referenced by the plugin is accessed by the server, returning contents which are then inserted into the form.

Field Container >

Field Section—Aligns fields into a column. Displays all field labels in an aligned column to the left of the fields. Fields can be divided into groups by inserting Headers and Instructions as needed. To display more than one column of fields, create a Form Columns container and place a Field Section container in each column.

Multicolumn Field Table—Organizes input fields into a multi-column table. It is recommended that you use Form Columns and Field Sections instead.

Entry Table with Field Instructions—Organizes input fields into a multicolumn table with fields on the left and instructions for each field on the right.

Component Template—Allows you to define a group of form elements that can be reused in more than one form. Changes to the template are propagated to all places where the template is used.

Tabs—Adds tabs to a form, each pointing to different content defined by a separate form.

Dynamic Menu—Displays a multicolumn menu based on data received from a document query to an adapter.

Form Columns—Divides the form into columns, allowing content to be grouped and organized.

Table >

Simple Table—Displays a list of documents resulting from a query.

Document Table—Displays a list of documents resulting from a query.

Tree—Displays a list of documents resulting from a query as a tree.

Portal Component >

Component Editor—Generates fields elements used to configure a specific portal component. Not intended for general Get-Resources use.

Portal Header—Generates the portal page header. Not intended for general Get-Resources use.

Corkboard Header—Generates header information needed by any page that includes a corkboard. Not intended for general Get-Resources use.

Corkboard Configurator—Generates a list of choices containing all known portal components. The list can be used to configure the components to display in a specific corkboard container.

Corkboard—Displays the portal components chosen and configured by each user.

Custom Configurator—Allows users to define their own custom component configurators.

Document Explorer >

> **Search**—Displays a personalized list of fields used to perform document searches.

> **List**—Displays a personalized table with the list of documents found as a result of a search.

> **Detail**—Displays a personalized view of a document detail.

Action >

> **Action**—Displays a button for an action. The button can be a link to another page or a submit action.

> **Default Action**—Defines a form's submit action when no actual buttons are displayed.

> **Back**—Navigates to the previous page of the Web application.

> **Home**—Navigates to the home page of the Web application.

> **Print**—Prints the current Get-Resources form.

> **Close**—Use to close pop-up windows.

Redirection—Redirects a page to a link depending on the result of the onload script matched against the condition

Transition—Contains an onload script and redirect arguments. After the script runs, execution is redirected according to the condition returned by the script. The options available from the Transition menu are the same as the Form menu, except there is no Action option.

- **Group of Strings**—List of multilingual strings.

  Multilingual String—The name of the StringResource is the ID of the string.

- **Group of Scripts**—Server-side ECMAScripts.

- **Script**—Server-side ECMAScript (JavaScript) file containing functions used by Web application forms.

  Header—Initial comments and imports required in this script file.

  Function—Script function defining application logic executed on the server. All functions that have public access should accept a Message object as the single input parameter and return a Message object as a response. For example:

```
function xyz( msg ) {var msgResponse=new Message(); ...
return msgResponse;}
```

A script requires this public access interface if it is used as an onload script for a form or if it is called directly via an Archway HTTP message.

- **Group of Scripts**—Server-side ECMAScripts.

- **Group of Triggers**—A collection of triggers. This container is not used by Get-Resources.

  - **Trigger**—Individual trigger for a document. This component is not used by Get-Resources.

    Message action—Message action executed by the trigger.

    Workflow action—Workflow action executed by the trigger.

    Script action—Script action executed by the trigger.

    Bizdoc Java action—Java action executed by the trigger inside Bizdoc.

  - **Group of Triggers**—Collection of triggers. This component is not used by Get-Resources.

    Trigger—Individual trigger for a document.

    Group of Triggers—Collection of triggers.

- **Group of Schemas**—Database schemas describing documents accessible by Get-Resources. Schemas define the field table mapping between Get-Resources and the back-end database.

  - **Raw Schema**—Description of a document's mapping on a real database.

  - **Schema**—not supported.

- **Group of Images**—Folder containing the image files to be used in your Web application.

  - **Image**—The image is loaded into the ImageData property as binary data. The file name property is used only the first time to load the image.

  - **Group of Images**—Folder containing image files.

    Image—The image is loaded into the ImageData property as binary data. The file name property is used only the first time to load the image.

    Group of Images—Folder containing image files.

- **Group of Presentation Files**—Folder containing files copied directly to the presentation folder for use within the Get-Resources Web server.

- **Text Presentation File**—Any generic file in the Presentation folder that is needed by the Web server, for example, client-side JavaScript, static JSP files.

- **Binary Presentation File**—Binary file outputted in the presentation folder. Accessed by the Web server and used by the browser.

- **Group of Presentation Files**—Folder containing files copied directly to the presentation folder for use within the Get-Resources Web server.

   Text Presentation File—Any generic file in the Presentation folder that is needed by the Web server, for example, client-side JavaScript, static JSP files.

   Binary Presentation File—Binary file outputted in the presentation folder. Accessed by the Web server and used by the browser.

   Group of Presentation Files—Folder containing files copied directly to the presentation folder for use within the Get-Resources Web server.

- **Group of default DocExplorer screens**—Folder containing default content for DocExplorer Personalization screens.

  - **Reference of a file**—File object.

  - **Directory Object**—not supported.

- **Group of Portal Components**—Components that appear in the portal components menu and can be added to the home page by the user.

  - **Portal Component**

     (**contents**)—The content of the portal component that is displayed.

     (**configure**)—Allows configuration of a portal component.

- **Group of Files**—A temporary container of miscellaneous files used by a Web application. For example, string files and scriptpoller.ini files are stored here.

  - **String file**—Temporary representation of a string file.

  - **Ini file**—Temporary representation of a scriptpoller.ini file.

- **Group of Strings**—List of multilingual strings.

  - **Multilingual String**—The name of the StringResource is the ID of the string.

- **Group of Roles**—not supported.

Group of Style Sheets > New >
- Style Sheet—Not supported.

Group of Roles—not supported.

Group of Files > New >
- **String file**—Temporary representation of a string file.
- **Ini file**—Temporary representation of a scriptpoller.ini file.

Group of Strings > New >
- **Multilingual string**—The name of the StringResource is the ID of the string.

Entities (collection of business objects) > New >
- **Entity**—This component is not used by Get-Resources.

Interfaces > New >
- **Interface**—Not supported.

System enumerations > New >
- **System enumeration**—Describes a system enumeration, used to define data attributes where the value stored is not the value displayed to the user. This allows multilingual databases.
  - **Value**—Defines one value for a system enumeration.

Templates > New

- **Schema** >Not supported.
- **Field Container**
  - **Component Template**

- **Directory Object**—not supported.
- **Group of Methods**—Includes a list of methods. You can create new methods under this element.
  - **Method**—Java Method. The name is not significant. You can add a comment to the method.

- **Method**—Java Method. The name is not significant. You can add a comment to the method.
- **Message action**— This component is not used by Get-Resources.

- **Workflow action**— This component is not used by Get-Resources.
- **Bizdoc Java action**— This component is not used by Get-Resources.
- **Script action**— This component is not used by Get-Resources.
- **Trigger**— This component is not used by Get-Resources.
- **Group of Images**—Allows you to create a group of images.
- **Attribute**— This component is not used by Get-Resources.
- **Reference**— This component is not used by Get-Resources.
- **Contain**—Contain an object as an embedded member.
- **Computed**—Computed property.
- **Structure**— This component is not used by Get-Resources.
- **Collection**— This component is not used by Get-Resources.
- **Methods**— This component is not used by Get-Resources.
- **Entity**— This component is not used by Get-Resources.

Types of form components

The following sections describe some of the more commonly used form components.

Component template containers

A component template is a special type of container used to store groups of preconfigured form components. A component template allows you to reuse the form components stored in the template throughout your project. After you create a component template, the component template name appears in the templates list of the Create and New menus. A component template references all the child form components and attributes settings defined in the template.

If you add a component template to Get-Resources and do not modify it, Peregrine Studio saves the form components as links to the component template. If you make changes to the form components in the template, Peregrine Studio saves only the changes you have made and links to the form components that you did not change.

Tip: Use component templates to re-use common elements of your forms. For example, if several of your forms contain customized search functionality, then you could create a component template that automatically calls the correct search schema, queries your back-end system, and displays the proper search fields.

To create a component template:

1 Right click the **Templates** nodes and click **New** > **Field Container** > **Component Template**.

Peregrine Studio adds a new component template node to the Project Explorer Tree.

2 Enter the name for the component template.

3 Right click the new component template node and use the **New** option to add form components.

4 Configure the form components you add to the template component. Peregrine Studio uses these settings as the default settings of the template component.

5 Save and build your Peregrine Studio project.

The new template component appears as an option in the New menu.

Important: Do not copy and paste or drag and drop items between template components. Instead add form components via the context-sensitive or Create menus. Studio does not use the linking features of template components on items that you copy from existing template components.

To add a component template to a form:

1 Right-click the form where you want the component template to be.

2 From the **New** menu, select the template you want to add.

Form components you can add to a component template

▶ All except Action and Redirection.

Tip: You can use a component template as the container for any form components that require a container. This is typically done for form components such as hiddenfields where you are not concerned about the display of the fields.

Attributes you can set for a component template

▶ Title, Summary, Order, User Role Restrictions, and Dynamic Runtime Restrictions.

Fieldsection containers

The fieldsection component is a container that aligns fields into a column. The fieldsection component displays each field on its own line in the column and aligns the field labels along the left of each field. Each fieldsection can have a border that surrounds the columns and visually indicates that the fields in the container are related. You can also add a header or instructions to your fieldsection as well as add labels and instructions to the individual fields in the fieldsection.

Tip: You can use the fieldsection form component to group and align related input fields. For example, if you have several fields to input search information, you can align the fields in a single fieldsection and add a header and instructions that will apply to all fields.

To create a fieldsection:

1 Right click the form where you want the fieldsection to be.

2 Click **New** > **Field Container** > **Field section**.

Form components you can add to a fieldsection

▶ Field, Component, HTML, Header, Import, and Instructions.

If you select the Header or Instructions form components, Studio will display the text editor screen for you to enter HTML code for your header and instructions. Peregrine Studio will not check the validity of your HTML code.

Attributes you can set for a fieldsection

▶ Title, Summary, Order, User Role Restrictions, Dynamic Runtime Restrictions, Border, and Readonly.

If you plan on having multiple fieldsections in a form, you can use the border Presentation property to display a line around a fieldsection to help visually distinguish the fieldsection from other elements in your Web application interface. You may also want to add a Form Columns layout container to display your fieldsections in two or more facing columns rather than a single column down the form.

Text edit fields

A text edit field provides a bordered field in which to display or enter a value as plain text. Text edit fields can only be added to forms within a container such as a component template or fieldsection.

The most common use for text edit fields is to provide a space for users to enter keyboard input. A text edit field saves the text entered into a particular schema field when a user submits the form.

Tip: To use a text edit field for text input, add an action to the form that submits the field information to another form. Set the Document Field attribute of the text edit field to the corresponding attribute name used in the document schema.

You can also use text edit fields to display information by default. To display information in a text edit field, create an onload server script that performs a document query, and then map the text edit field to one of fields of the schema.

Tip: To use a text edit field to display of information by default, add a schema to the parent form that defines the information to be displayed. Set the Document Field attribute of the text edit field to the corresponding attribute name used in the schema. Set the readonly attribute under Presentation to Yes if you do not want users to change the information displayed.

To create a text edit field:

1 Right-click the container where you want the field to be. This displays the context-sensitive menu.

2 Click **New** > **Field** > **Text Edit**.

Form components you can add to a text edit field

▶ None.

Attributes you can set for a text edit field

▶ Instructions, Label, Title, Document Field, Display Value, Max Characters, and Data.

Selectbox fields

A selectbox provides a drop-down list box from which users can select predefined values. You can add items to the selectbox in one of two ways:

- Explicitly define the options. The selectbox always displays the options you enter and always displays them in the order you define them in the Order attribute.

- Query your back-end database and generate an XML document that provides the display options. The selectbox displays the options as defined by the schema used to generate the XML document. Typically, the selectbox uses the same schema as the form of which it is a part. If you want to use a schema to display the options in a selectbox, then you must set the Document field attribute to an attribute name in a schema.

Tip: Use the schema query method to avoid duplicating information that is already stored in your back-end database. If you explicitly enter the options in the selectbox, then you have to update, rebuild, and re-deploy your project every time you change the list of selectbox options. If you store the selectbox options on your database, however, then you only need to change the database values, and your schema query will automatically pick up any changes you make.

When you are working with selectboxes, keep in mind that:

- You can only add selectbox fields within a container such as a component template or fieldsection.

- Users cannot add entries to selectbox fields. To implement such functionality, you would need to write a client-side JavaScript to insert any information added into your back-end databases.

- Get-Resources uses selectbox fields to constrain user input to a list of predefined items. The selectbox field saves the selected item to a particular field when a user submits the form. The field used to save the information must match a field defined in a document schema.

- If you have a large number of selections for users to choose from you may want to use a lookupfield in place of a selectbox. The advantage of using lookupfields are:

  - they can be personalized
  - they are not loaded into memory until the lookupfield is selected, which reduces the amount of time necessary to render the form.

To create a selectbox field:

1 Right click the container where you want the field to be.

2 Click **New** > **Field** > **Selectbox**.

Form components you can add to a selectbox field

▶ Option. The Option form component allows you to explicitly define the entries displayed in the selectbox.

Attribute categories you can set for a selectbox field

▶ Instructions, Label, Title, Document Field, Display Value, Size, Multiple Selection, Permit Blank, Data, Presentation, Events, User Role Restrictions, Dynamic Runtime Restrictions, Process, Presentation, and Databound.

Databound attributes

The Databound attributes are where you will define what schema and schema attributes provide the information for the selectbox. The following list describes what information to enter in the Databound attributes.

- Document. Enter the schema name you want to use to query and display the information requested in the selectbox.

- Values. Enter the attribute name from your schema that defines what information you want to use to sort and identify the information in the selectbox. This value can be identical to the displaylist attribute, but it is recommended that you use the Id attribute name defined in the schema. The Id attribute is the preferred choice because it is a unique value and requires less memory to sort since it is only a number.

- Captions. Enter the attribute name from your schema that defines what database information you want displayed in the selectbox.

Hidden data fields

A hidden data field stores form information without displaying it to the user. Get-Resources fopasses the information stored in a hidden data to other forms when the form is submitted.

Tip: You can use hidden data fields to prevent users from having to input the same information on multiple forms. For example, if a user enters contact information in one form, then you can use hidden data fields to store this contact information in later forms.

To create a hidden data field:

1 Right click the container where you want the field to be.

2 Click **New** > **Field** > **Hidden Data field**.

Form components you can add to a hidden data field

▶ None.

Attributes you can set for a hidden data field

▶ Document Field, Display Value, Visible Flag, Unique Key Field, User Role Restrictions, and Dynamic Runtime Restrictions.

Redirections

A redirection takes users to another form when the onload server script generates a certain condition. A conditional redirection requires the parent form to run a server script when it is loaded. To use a conditional redirection, you must create a server script that checks for a particular condition and then outputs a condition message when this condition occurs.

You can only add a redirection to a form; you cannot add a redirection to a form component.

Tip: You can use a redirection to take users to a form when they enter particular information or a particular result, such as when an error occurs or when no results are generated.

To create a redirection:

1 Right-click the form where you want the redirection to be.

The context-sensitive menu is displayed.

2 Click **New** > **Redirection**.

Form components you can add to a redirection

▶ None.

Attribute categories you can set for a redirection

▶ Visible flag, Condition, Frameset, HTTP Submit Method, Parameters, Target (form, field, or URL), User Role Restrictions, and Dynamic Runtime Restrictions.

Redirection attributes

For most redirections, the two most important attributes to set are the condition and the target form.

- Condition. Enter the message generated by your server script that activates the redirection to another form. If there is no condition, the redirection will activate every time the page is loaded. See *Common message operations* on page 18 for more information on setting a condition.

- Target form. Enter the full Peregrine Studio path to the form where the user should be redirected.

Simple table

A simple table is a container to display information generated from a schema document query. The simple table form component only has two basic functions by itself. The simple table form:

- Calls the schema that will generate the table data, and
- Describes how the data will be displayed in the columns of the table.

A simple table requires columns components in order to display data.

To create a simple table:

1 Right-click the form where you want the table to be.

2 Click **New** > **Table** > **Simple Table**.

Form components you can add to a simple table

▶ Link, Text Column, Entry Column, Spinner Column, Select Column, Radio Button Column, Checkbox Column, Image Column, Link Column, and Lookup Column.

Attributes you can set for a simple table

▶ Visible Flag, Caption (en), Accessibility Title (en), Accessibility Summary (en), Size, Preview, Order, Readonly, Required, Column Sorting, Border, Process, Document, Data, Dynamic Headers and Columns, Instructions (en), Events, User Role Restrictions, and Dynamic Runtime Restrictions.

The Document attribute defines the schema the simple table uses. You can enter a schema name or select one from the drop-down list box.

Simple tables include a built interface to view large tables in smaller pages. You can use the size attribute to set the number of rows to display on one page. When users want to view more of the table results, they can click on the next *x* rows button to view the next page of table rows. All simple tables include the link icons to browse forward and backward in the table.

Document table

A document table is a container you can use to display any other form component from within a table. The document table form component only has two basic functions by itself. The document table form:

- Calls the schema that will generate the table data, and
- Describes how the data will be displayed in the columns of the table.

A document table requires columns components in order to display data. Unlike the simple table, the document table only uses one type of column: the formcolumn. However the formcolumn form component allows you to add any other form component to your document table.

To create a document table:

1 Right-click the form where you want the document table to be.

2 Click **New** > **Table** > **Document Table**.

Form components you can add to a document table

▶ Column.

Attributes you can set for a simple table

▶ Visible Flag, Accessibility Title (en), Accessibility Summary (en), Size, Preview, Order, Border, Document, User Role Restrictions, and Dynamic Runtime Restrictions.

The Document attribute defines the schema the document table uses. You can enter a schema name or select one from the drop-down list box.

Document tables include a built interface to view large tables in smaller pages. You can use the size attribute to set the number of rows to display on one page. When users want to view more of the table results, they can click on the next x rows button to view the next page of table rows. All document tables include the link icons to browse forward and backward in the table.

Table links

A table link allows the user to click on a table row and be redirected to another form. The table link also saves some field information about the row the user selects and submits this information to the target form. Table links are typically used for two functions:

- To display more information about an item selected in the table, or
- To copy certain information about the item selected in the table into a new form such as, for example, the price of an item in a purchase request form.

To create a table link:

1 Right-click the table where you want the table link to be.

2 Click **New** > **Link** > **Table Link**.

Form components you can add to a table link

▶ None.

Attributes you can set for a simple table

▶ Visible Flag, Label (en), Title (en), Balloon (en), Style Class, Data, Image, HTTP Submit Method, Parameters, Target (frame, form, field, script, or URL), Events, User Role Restrictions, and Dynamic Runtime Restrictions.

Table link attributes

For most table links, the two most important attributes to set are the Document field and the target form.

- Document field. Enter the field that describes what information should be passed when a table link is submitted. The Document Field attribute should match the attribute name of an item in your schema. The attribute is typically set to the Id schema attribute.

- Target form. Enter the full Peregrine Studio path to the form where the user should be redirected when they click on a table row.

Text columns

A text column displays the results of a document query in a table column as plain text. Each text column displays one field of information from a back-end database. The field must match an attribute name listed in the document schema of the parent table.

When working with text columns, keep in mind that they:

- Are always read-only and cannot be used to update information in the back-end database.
- Can only be added as child nodes of a simple table.

To create a text column:

1 Right click the table where you want the text column to be.

2 Click **New** > **Text Column**.

Form components you can add to a text column

▶ None.

Attributes you can set for a text column

▶ Visible Flag, Order, Label (en), Title (en), Support Links, Data Type, Document Field, Translation Function, Style Class, Events, User Role Restrictions, and Dynamic Runtime Restrictions.

Text column attributes

For most text columns, the two most important attributes to set are the Document field and the Label (en).

- Document Field. Enter the field that describes what information should be displayed in the text column. The Document Field attribute should match the attribute name of an item in your schema.

- Label (en). Enter the label you want displayed in the first row of the table as the column heading. If you are using dynamic headers and columns, you will want to leave this attribute blank.

Form columns

A form column is a container for any other form component you want to add to a table. Unlike a text column, a form column can display any number of fields of information from your back-end database. Each field, however, must still match an attribute name listed in the document schema of the parent table.

When working with form columns, keep in mind that they:

- Can contain form components that insert or update information in your back-end database.

- Can only be added as child nodes of a document table.

To create a form column:

1 Right click the document table where you want the text column to be.

2 Click **New** > **Column**.

Form components you can add to a form column

▶ Any.

Attributes you can set for a form column

▶ Visible Flag, Label (en), Order, User Role Restrictions, and Dynamic Runtime Restrictions.

Form column attributes

For most form columns, the two most important attributes to set are the Label (en) and the User Role Restrictions.

- Label (en). Enter the label you want displayed in the first row of the table as the column heading. If you are using dynamic headers and columns, leave this attribute blank.

- User Role Restrictions. Enter the user role or roles that you want to have access this form column. Only users with this access level will have access to the form components in the column. If you do not want to set a user role restriction, leave this attribute blank.

Actions

An action is a button that submits form information or follows a particular link. The following is a list of the possible actions you can include in your forms:

- Action. Use to submit form information or follow a link.
- Back. Use to navigate back to the previous form.
- Close. Use to close pop-up windows.
- Default Action. Use to define a form's submit action when no buttons are displayed in a form.
- Home. Use to navigate to the portal home page.
- Print. Use to print the current form.

To create an action:

1 Right-click the form where you want the action to be.

2 Click **New** > **Action** and then click the action type you want to add.

Form components you can add to an action

▶ None.

Attributes you can set for an action

▶ Submit Form, Target (frame, form, field, script, or URL), Label (en), Title (en), Balloon (en), Image, Parameter, HTTP Submit Method, Events, User Role Restrictions, Dynamic Runtime Restrictions, Visible Flag, and Presentation.

Action attributes

For most actions, the three most important attributes to set are the Image Folder, Target form and the Label (en).

- Image. Enter the file name of the image to be used for the button.
- Target form. Enter the full Peregrine Studio path to the form where the user should be redirected when they click on the button.
- Label (en). Enter the label you want displayed in the button.

5 Scripting

CHAPTER

This chapter provides an overview of how scripts are put together and used. You should be familiar with JavaScript and ECMAScript and should have access to the JavaDocs provided with your Get-Resources installation.

This chapter covers the following topics:

How scripts are used

Get-Resources uses scripts to query back-end databases and to format the results into XML documents based on schemas. Generally, you will only need to create new scripts if you create new forms. Most customizations do not require changes to the script, but rather to the schema that the script uses to display data. When you need to create or make changes to a script, you must have created or activated a writable package extension in which to save your changes.

Tip: You can use the existing scripts as templates for your custom scripts. Try and find a script that has similar functionality to what you want, and then copy and paste the script into your Peregrine Studio project.

Types of scripts

Get-Resources uses two types of scripting to transfer and format data between your back-end databases and Web application forms:

- Server-side scripting—Server-side scripts run from a Web server. Server-side scripts have access to both user-submitted form data and any data generated by a back-end system. The output of server-side scripts can be returned to both a back-end system and the remote browser. All Get-Resources server-side scripts are written in ECMAScript. An example of server-side scripting would be querying a back-end system for the list of items associated with a particular order.

- Client-side scripting—Client-side scripting runs from a JavaScript-capable browser. Client-side scripts have access to user data before it is submitted to a Web server and any back-end data that was uploaded with the current Web page. The output of client-side scripts can be used only by the client browser. All Get-Resources client-side scripts are written in JavaScript. An example of using client-side scripting would be updating the total price displayed on an order form when an amount is entered in another field of the page.

Where scripts are stored

The following table describes how you can include both types of scripting into your projects.

Script type	Language used	Where created and stored
Server-side	ECMAScript	You can author server-side scripts only in Peregrine Studio. Each script then becomes an object available for use throughout the project.
Client-side	JavaScript	You can author client-side scripts outside of Peregrine Studio and add them to your project. You can also include client-side scripts as part of the HTML code stored with a form.

Peregrine Studio stores all server-side ECMAScripts as part of your project file. At build time, Peregrine Studio copies the scripts into your application server's deployment folder and creates all necessary Get-Resources JSP pages. At run time, the deployment application server executes the JSP pages along with any server-side scripts called by the JSP pages and sends the output to the client browser. The client browser will execute any client-side JavaScript present in the rendered JSP page.

How scripts are used

The Archway servlet supports several different methods to invoke and utilize scripts within Get-Resources. The following sections describe the different ways in which ECMAScript and JavaScript can be used within Get-Resources.
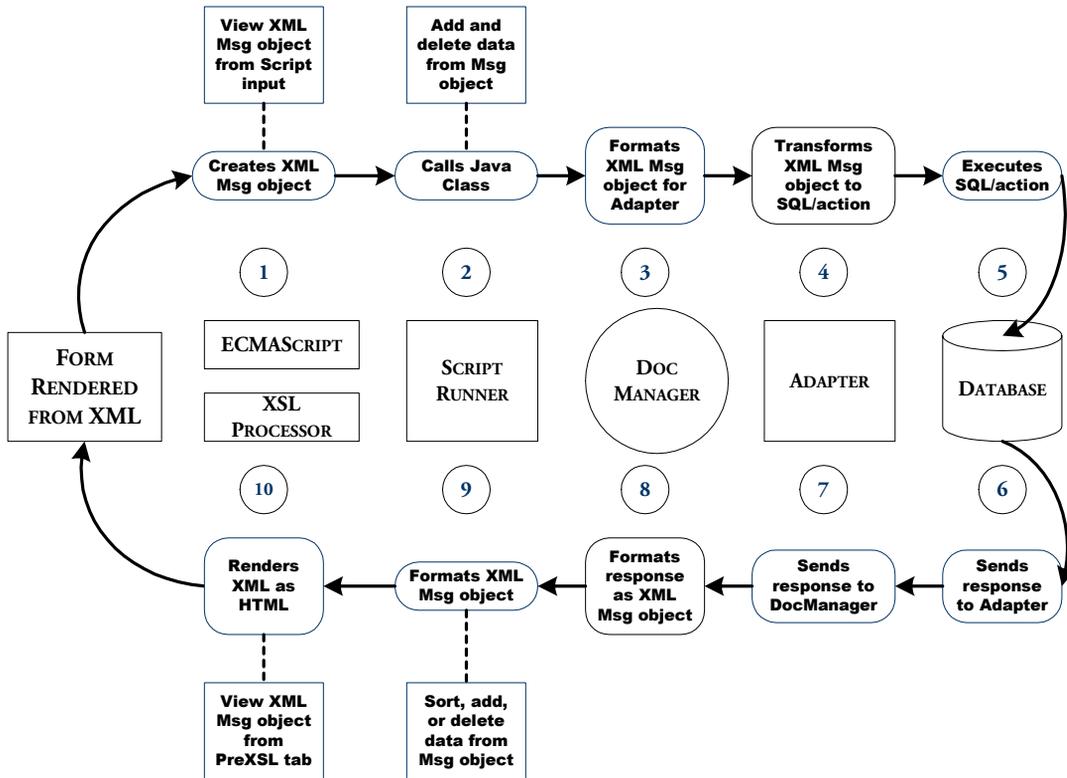
Forms—Server Side

All Get-Resources forms support invoking onload server-side scripts. Typically, the onload script creates an XML message to gather and format information from a back-end database. The script message can contain queries or updates to the database or to XML documents built from a schema. The scripts typically use a schema, one or more input parameters, and a back-end database query to create an XML document.

Many server onload scripts use one of the following API calls:

- sendDocQuery—sends an SQL or XML document query to the back-end database. Archway queries the record using the table and field information supplied by the schema. The database then returns the results of the query as an XML document formatted as defined in the schema.

- sendDocInsert—sends an XML document to the back-end database that describes a new record. Archway creates the new record in the database using the table and field information supplied by the schema.

- sendDocUpdate—sends an XML document to the back-end database that describes an update to an existing database record. Archway updates the record using the table and field information supplied by the schema.

- sendDocDelete—sends an XML document to the back-end database that describes a record in the database to be deleted. Archway deletes the record using the table and field information supplied by the schema.

Get-Resources typically use the following ECMAScript syntax to refer to schemas. For additional methods of formatting these messages, refer to the JavaDocs API documentation provided with your Get-Resources installation.

```
archway.sendDocQuery( "adapter name", "schema name", input msg);
archway.sendDocInsert( "adapter name", message object);
archway.sendDocUpdate( "adapter name", message object);
archway.sendDocDelete( "adapter name", message object);
```

- For *adapter name*, enter the name for the back-end database adapter. The adapter listed here will use the ODBC connection that you have defined in the achway.ini file. For most applications, the adapter will be a two letter name.

- For *schema name*, enter the name defined in the <document name="schema name"> element of the schema file.

- For the *input msg*, enter the variable name of a message that OAA uses to store input parameters for the ECMAScript function. The default input message is the msg object that is defined in all onload functions. The input message is the XML message containing the HTML page parameters.

- For *message object*, enter a variable name of a message object containing a schema name and any input parameters.

For example, the script sample below defines a variable called *msgReturn* that sends a document query to ServiceCenter using the empdetail schema and any input parameters stored in the *msg* message object. The variable *msgReturn* then returns the result of the document query.

```
var msgReturn = archway.sendDocQuery( "sc", "empdetail", msg );
return msgReturn;
```

Client Side

The browser handles all client-side scripting when a user views a Web application.

Note: Peregrine does not provide customer support for custom client-side scripts.

Testing scripts

Get-Resources offers two means of testing your ECMAScript:

- Rhino JavaScript Debugger
- URL Queries

Rhino JavaScript debugger

You can now configure Get-Resources to send script output to the Rhino JavaScript Debugger provided by Mozilla. The Rhino JavaScript Debugger provides a graphical user interface for debugging interpreted JavaScript and ECMAScript. When you enable the Rhino JavaScript Debugger, you can log on to the Get-Resources server and see debugging information about your installation as you browse through the Get-Resources interface.

Important: To use the Rhino JavaScript debugger your application server cannot be configured to run as a service.

To enable the Rhino JavaScript debugger:

1 Login to the Get-Resources administration page.

2 Click **Settings** > **Logging** tab.

3 For the Debug script option, select **Yes**.

4 Click **Save** to store your changes.

5 Login to the Get-Resources server.

6 Browse to the Get-Resources deployment directory. By default this directory has the following path:

`<application server>\<context>\WEB-INF`

For *<application server>*, enter the installation path to your application server. For example, `C:\Program Files\Peregrine\Common\Tomcat4`

For *<context>*, enter the path where you deployed the Get-Resources files. For example, `webapps\oaa`.

7 Using any text editor, open the file `local.xml`.

8 Add the following line anywhere between the `<settings>` elements:

```
<showDebugger>true</showDebugger>
```

9 Save the file.

10 Copy the file rhinodebugger.jar from the Get-Resources Tailoring Kit Installation CD to the following path on your test server:

<application server>\<context>\WEB-INF\lib

For *<application server>*, enter the installation path to your application server.For example, C:\Program Files\Peregrine\Common\Tomcat4

For *<context>*, enter the path where you deployed Get-Resources the files. For example, webapps\oaa.

11 Restart your application server.

The Rhino JavaScript Debugger appears the next time you start your application server on this system.
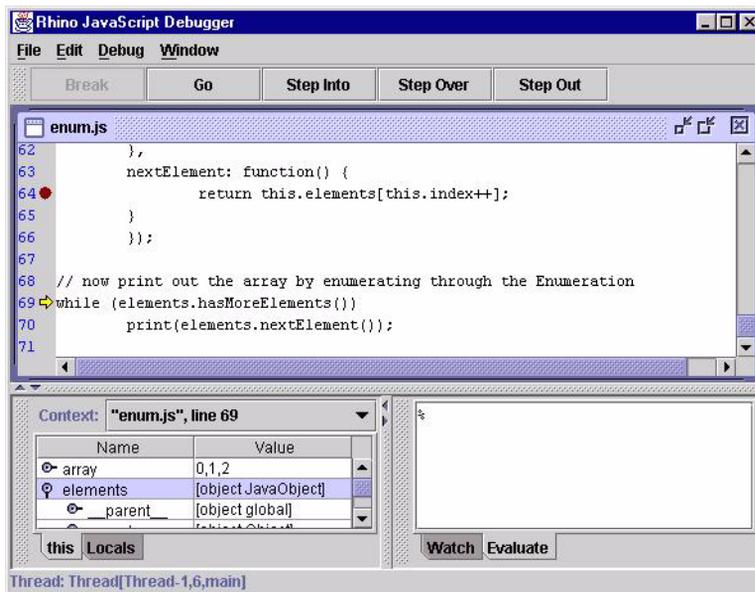
For more information about the Rhino JavaScript Debugger, see the Mozilla Web site:

http://www.mozilla.org/rhino/debugger.html

URL queries

You can test the output generated by your server-side onload scripts and schemas by using URL queries to the Archway servlet.

Archway will invoke the server script or schema as an administrative user and return the output as an XML document. Your browser will need an XML renderer to display the output of the XML message.

Using URL queries can be useful for debugging your tailoring changes and for using the Archway servlet without having to log into Get-Resources.

Note: Your browser may prompt you to save the XML output of the URL query to an external file.

URL Script Queries

Archway URL script queries use the following format:

```
http://server name/oaa/servlet/archway?script name.function name
```

- For *server name*, enter the name of the Java-enabled Web server. If you are testing the script from the computer running the Web server, you can use the variable localhost as the server name.

  The **/oaa/servlet** mapping assumes that you are using the default URL mapping that Get-Resources automatically defines for the Archway servlet. If you have defined another URL mapping, replace the servlet mapping with the appropriate mapping name.

- For *script name*, enter the name of the script you want to run.
- For *function name*, enter the name of the function used by the script.

URL Schema Queries

Archway URL schema queries use the following format:

```
http://server name/oaa/servlet/archway?adapter name.Querydoc
&_document=schema name
```

- For *adapter name*, enter the name for the back-end database adapter the schema uses. The adapter listed here will use the ODBC connection that you have defined in the Admin module Settings page.

- For *schema name*, enter the name defined in the <document name="schema name"> element of the schema file.

  The **/oaa/servlet** mapping assumes that you are using the default URL mapping that Get-Resources automatically defines for the Archway servlet. If you have defined another URL mapping, replace the servlet mapping with the appropriate mapping name.

Your script output should be similar to this.
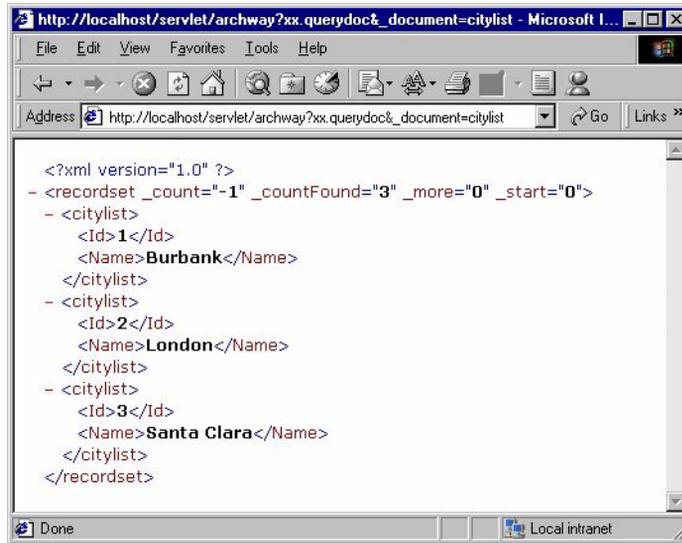
URL SQL Queries

Achway URL SQL queries use the following format:

```
http://server name/oaa/servlet/archway?adapter name.query&_table=
table name&field name=value&_[optional]=value
```

- For *adapter name*, enter the name for the back-end database adapter the schema uses. The adapter listed here will use the ODBC connection that you have defined in the Admin module Settings page.

- For *table name*, enter the SQL name of the table you want to query from the back-end database.

- For *field name*, enter the SQL name of the field you want to query from the back-end database.

- For *value*, enter the value you want to the field or optional parameter to have.

- For *_[optional]*, enter any optional parameters to limit your query. Examples include:

    - _return. Returns the values only of the fields you list.

    - _count. Specifies how many records you want returned with the query.

The **/oaa/servlet** mapping assumes that you are using the default URL mapping that Get-Resources automatically defines for the Archway servlet. If you have defined another URL mapping, replace the servlet mapping with the appropriate mapping name.

Common message operations

The following section describes some common methods that server-side scripts can be used to create XML messages. Refer to the JavaDocs (especially, com.peregrine.oaa.core.Message) for more information about and examples of XML message operations.

- Create a new generic message. You can use archway.sendDocQuery() to create a generic XML message. You can then add elements to the XML message with other methods.

  ```
  var msgQuery = new Message();
  ```

  Creates an empty XML message called msgQuery.

- Create a new message with a specific XML element tag. You can then use archway.sendDocUpdate() and archway.sendDocInsert() to send the XML message to the back-end database.

  ```
  var msgRequest = new Message( "Request" );
  ```

  Creates an XML message called msgRequest with the element <Request>.

- Add a value to a particular XML element. You can use this method to add a new element and value to the XML message.

  ```
  msgQuery.add( "LastName", "Jones" );
  ```

  Adds the value Jones to the element <LastName>. The output is in standard XML format: <LastName>Jones</LastName>.

- Set the value of an XML element. You can use this method to overwrite the value of an existing element in the XML message.

  ```
  msgQuery.set( "LastName", "Jones" );
  ```

  Sets the value of the element <LastName> to Jones. The output is in standard XML format: <LastName>Jones</LastName>.

- Get the value of an element in the XML message. This method returns an empty string "" if there is no value for the element.

  ```
  var strName = msg.get( "LastName" );
  ```

  Sets the variable strName to the value of the element <LastName> in the XML message. For example, if the XML message contains the element <LastName>Jones</LastName> then strName uses the value Jones.

- Get all of the elements and values (the subdocument) listed under a particular element in the XML message. This method returns an empty string "" if there is no subdocument for the element.

  ```
  var msgRequest = msg.getMessage( "Request" );
  ```

Sets the variable `msgRequest` to the subdocument listed under the element
`<Request>` in the XML message. For example, suppose the XML message
contains the following elements:

```
<Request>
  <ID>1234</ID>
  <LastName>Jones</LastName>
  <Status>Approval</Status>
</Request>
```

Then, the msgRequest uses the subdocument:

<ID>1234</ID><LastName>Jones</LastName><Status>Approval</Status>.

■ Set a script condition when the script returns a particular XML message
result. You can use conditions to control when Peregrine Studio form
components such as redirections and access fields should be activated. For
example, the following script checks the value of the `Name` element:

```
if ( msg.get( "Name" ) == "" )
{
msgResponse.setCondition( "error" );
  return msgResponse;
}
```

This function searches the XML message for the value of the `<Name>`
element. If the value is empty, then the script sets the `error` condition.

■ Return the number of instances that a particular element appears in an
XML message. You can use this method to set a condition for further
actions. For example, the following script uses the `getList` method to set a
condition:

```
var list = msgResponse.getList( "Location" );
if ( list.getLength() == 0 )
msg.setCondition( "noresults" )
var i = 0;
for ( i = 0;i < list.getLength(); i++ )
{
// add function to process records in the list ...
}
```

Sets the variable `list` to the number of `<Location>` elements in the XML
message. If the number of instances is zero, then the script sets the `noresults`
condition, otherwise the script performs some other action.

■ Log the contents of a particular XML message. This method saves the
output of the script to the file `archway.log`. This is another way of
debugging your ECMAScript in addition to the *Rhino JavaScript debugger*
on page 85.

- Using a logging domain. You can use a logging domain to group log messages from a particular component or script.

```
env.debuglog( "Get-Resources", "sendDocQuery returned the
message ", msgResponse );
```

- Without using a logging domain

```
env.debuglog( "sendDocQuery returned the message ",
msgResponse );
```

Important: You must enable the Debug Logging option from the Get-Resources administrative interface (**Administration** > **Settings** > **Logging tab**).

Tip: Remove or comment out this method before deploying to your production environment as script logging is CPU-intensive and degrades server performance.

Using ECMAScript in an object oriented manner

ECMAScript implementation in Get-Resources

All Scripts defined in Peregrine Studio end up being loaded as one ECMA script object. The functions defined in the Script are the object's methods, and the variables declared outside a function are the object's attributes. This implementation as an object is what enables you to use the **dot** syntax to call scripts and functions.

Name resolution in ECMAScript

Every ECMAScript object has a special property: its *prototype*. A prototype is an ECMA script object, and it is used in the property name resolution for the object.

Every script is run within a scope that holds a set of objects and variables declared in the same scope.

When you access a property or call a function in a given environment, ECMA script tries to resolve the name in the current scope first (usually the function's context). If it does not find it, it tries in the current object's prototype. If it does not find the property in the prototype, or in the prototype's prototype, the ECMAScript engine searches in the parent scope.
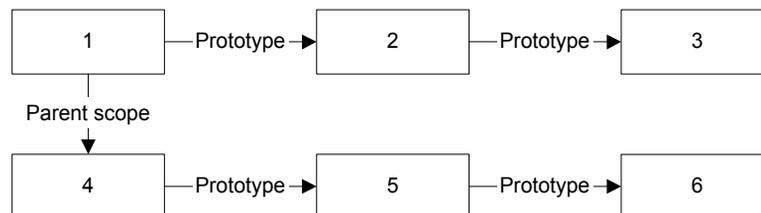
Using the object prototype for object oriented programming

The fact that ECMAScript looks up for a variable name or a function name in the prototype if it does not find it in the object, gives some ability to define a standard behavior as an object's method, and make this object the prototype for another object that can overwrite the behavior by providing a method with the same name.

The following is an example that you can try with the ECMAScript command line utility.

To use an object prototype:

1 In the **WEB-INF/lib** folder, type **java -jar js.jar** to start the command line.

```
function vehicle()
{
  function _start ()
  {
    print("starting " + this.getVehicleName())
  }
  this.start = _start;
  this.getVehicleName = new Function("return 'vehicle'; ");
}

function airplane()
{
  this.getVehicleName = new Function("return 'airplane'; ");
}
airplane.prototype = new vehicle();


function car(make)
{
  this.getVehicleName = new Function("return 'car ' + this.make;");
  this.make = make;
}
car.prototype = new vehicle();
```

2 Create three objects, one for each class:

```
var myVehicle = new vehicle();
var myPlane = new airplane();
var myHonda = new car("Honda");
```

3 Try the start method for each of these objects:

```
js> myVehicle.start()
starting vehicle

js> myPlane.start()
starting airplane

js> myHonda.start()
starting car Honda
```

You can see that although the airplane **class** and the car **class** do not implement the start method, start is found in their prototype. You can also see that since these two **classes** overwrite the **getVehicleName** function, the start method calls the method that was defined in the object. These are standard behaviors in object-oriented languages.

Overwriting a method to extend the parent class method can be more complicated in ECMA script.

To overwrite a method to extend the parent class method:

1 Create a sports car class that derives from the car class, and extend the start method to add a warm-up phase before the car actually starts.

```
function sportscar1(make)
{
  // other way to declare that the prototype for the
  // sportscar object is a car object. Contrary to
  // the other way, where only one vehicle object is the
  // prototype of all the car objects, here there will be
  // one car object per sportscar1 object.
  this.parentCar = new car(make);
  this.__proto__ = this.parentCar;
  // Extend the start function
  function _start()
  {
    print("warming up");
    this.parentCar.start();
  }
  this.start = _start;
  // Change also the vehicle name to reflect that this is
  //a sports car
  this.getVehicleName = new Function("return 'sports car ' +
this.make;");
}
```

2 Create an object for this class:

```
var myMaserati = new sportscar1("Maserati");
```

3 Call the start method:

```
js> myMaserati.start();
warming up
starting car Maserati
```

You can see that the new start method is called, that the start method declared in vehicle is called as well. But the new getVehicleName was not called, as the second line that was printed should show as **starting sports car Maserati**. This is because using this.parentcar.start() changes the scope in which the start function is called from the sportscar1 object to the parentcar object (car class), and as a result the getVehicleName is resolved in the scope of the car object. To change this behavior, a the parent function must be called in a specific way that is illustrated in the following sportscar2 class.

```
function sportscar2(make)
{
  this.parentCar = new car(make);
  this.__proto__ = this.parentCar;
  // Extend the start function
  function _start()
  {
    print("warming up");
    this.parentCar.start.apply(this, arguments);
  }
  this.start = _start;
  // Change also the vehicle name to reflect that this is
  //a sports car
  this.getVehicleName = new Function("return 'sports car ' +
this.make;");
}
```

To change the start method:

1 Create an object for this class:

   var myFerrari = new sportscar2("Ferrari");

2 Call the start method:

```
js> myFerrari.start();
warming up
starting sports car Ferrari
```

You can see that we now get the expected result. The code is using the apply method of the Function object, and passes the object that will be used as this first, and the arguments that were passed to the current function (_start).

Note: The code uses this.parentCar instead of this.__proto__, which could seem to be valid, but can cause an infinite recursive call if another class deriving from sportscar2 extends the start method and calls it parent, because this.__proto__ would still be evaluated against the derived object, and the start function in sportscar2 would keep calling itself. It is therefore preferable to store the parent object in a variable that is not overwritten by the subclasses. Here, with a nomenclature that uses the **parent** prefix and the parent class name, the uniqueness is ensured. You can try if you want with a racecar class that would derive from sportscar2 and overwrite the start function by calling the parent)

How to use object orientation for tailoring

In Get-Resources, objects are instantiated automatically from the script files when they are loaded in memory. To implement the prototype hierarchy, the __proto__ attribute must be set in a script file's header.

For example:

```
import requestinterfacebase;
this.__proto__ = requestinterfacebase.valueOf();
```

The previous valueOf method returns a pointer to the requestinterfacebase object. The line is equivalent to this.__proto__ = requestinterfacebase;.

If you need to call a parent method, you can specify it using the dot format. For example:

```
// Submit the request (Call the parent method)
var msgNewRequest = requestinterfacebase.saveRequest.apply
(this, arguments);
```

As long as each object has a unique name, for example the script name, there is no need to store the parent object in a member variable. In that respect, using object orientation in Get-Resources is simpler than in the general case.

Sample scripts

The following sections provide sample server-side ECMAScripts and descriptions that you can use as templates in Get-Resources. If you need help with a client-side scripting, a list of suggested reference materials is provided on page 105.

General script samples

You can use ECMAScript to serve a number of different functions such as creating an XML document from a schema, running a SQL query, or formatting the data received from a database query. The following samples show some of the ways in which you can use ECMAScript to gather data.

Selecting a field from a schema

```
function getCityList ( msg )
{
//Query sample database for the records using the citylist
//schema
var msgQuery=newMessage();
msgQuery.set("_return", "Name");
var msgReturn=archway.sendDocQuery ("xx","citylist", msgQuery);

return msgReturn;
}
```

Input

A message object, msg. This script does not typically have input from any previous form. If you change this script to be part of a results form, then the input message could contain form fields or values from a prior list form.

Output

The script produces an XML document built from the schema and adapter specified in the sendDocQuery function. The XML output below is an example of the kind of data that could be returned using a similar script.

```
<recordset _count="-1" _countFound="3" _more="0" _start="0">
  <citylist>
    <Id>1</Id>
    <Name>Burbank</Name>
  </citylist>
  <citylist>
    <Id>2</Id>
    <Name>London</Name>
```

```
    </citylist>
    <citylist>
      <Id>3</Id>
      <Name>Santa Clara</Name>
    </citylist>
</recordset>
```

Although the sendDocQuery function specifies only the <Name> element,
Archway automatically includes the <ID> element in the XML document
produced. This is expected behavior of the Archway servlet.

Description

This script gathers a list of city names for an employee search form. The
sendDocQuery function creates an XML document built from the **citylist**
schema and searches for the value of the <Name> element. You can use
parameters like "Name" in your script messages to limit or add to the list of
values returned by your schema query.

Calling other scripts and combining the results

```
function getSearchInfo( msg )
{
//Create empty variable msgResponse
var msgResponse = new Message();

//Call getDepList function and add results to msgResponse.
msgResponse.add( this.getDepList( msg ) );
// Call getCityList function and add results to msgResponse
msgResponse.add( this.getCityList( msg ) );

return msgResponse;
}
```

Input

A message object, msg. This script does not typically have input from any previous form. If you change this script to be part of a results form, then the input message could contain form fields or values from a prior list form.

Output

The script produces an XML document built from two other scripts, getDepList and getCityList. Each script adds to the XML document stored in the msgResponse variable by running a sendDocQuery function with a schema. The XML output below is an example of the kind of data that could be returned using a similar script.

```
<_doc>
<recordset _count="-1" _countFound="19" _more="0" _start="0">
  <departmentlist>
    <Id>1</Id>
    <DepartmentName/>
  </departmentlist>
  <departmentlist>
    <Id>2</Id>
    <DepartmentName>Administration</DepartmentName>
  </departmentlist>
  <departmentlist>
    <Id>3</Id>
    <DepartmentName>Administrative Services</DepartmentName>
  </departmentlist>
  <departmentlist>
    <Id>4</Id>
    <DepartmentName>Burbank Agency</DepartmentName>
  </departmentlist>
...
```

```
</recordset>
<recordset _count="-1" _countFound="3" _more="0" _start="0">
  <citylist>
    <Id>1</Id>
    <Name>Burbank</Name>
  </citylist>
  <citylist>
    <Id>2</Id>
    <Name>London</Name>
  </citylist>
  <citylist>
    <Id>3</Id>
    <Name>Santa Clara</Name>
  </citylist>
</recordset>
<_form>e_employeelookup_search_search.jsp</_form>
</_doc>
```

Description

This script generates the city and department names that a user can select from in an employee search form. The .add function appends the output of the **getDepList** and **getCityList** functions to the msgResponse variable. The two script references use the relative naming convention (this) to indicate that the functions called are part of the same script as **getSearchInfo**.

Form script sample

Most ECMAScripts run during a form's onload processing. Typically, form scripts query and format data for display in a Web application form, but you can also use them to update existing database records or insert new ones. The following samples show how to use server onload scripts to search a database for employee information.

Creating an XML document from a schema

```
function getEmpList( msg )
{
//Add Department subdocument to the input message
var strReturn = msg.get("_return");
if ( strReturn.length > 0 )
  msg.set("_return", strReturn + ";Department");

//In msg, set sort to LastName and then FirstName
msg.add( "_sort", "LastName,FirstName" );

//Query sample database for the records using the
//employeedetail schema and the criteria found in the msg object
var msgReturn = archway.sendDocQuery( "xx", "employeedetail", msg );
//Test if the number of items returned is zero, if true set
//ListEmpty condition
if ( msgReturn.get("_countFound") == "0" )
  msgReturn.setCondition( "ListEmpty" );

//Return the contents of the msgReturn variable
return msgReturn;
}
```

Input

A message object, msg. This script has an input message from a previous search form. In this case, the input message is amended to include a subdocument, Department, in addition to any other input data passed to the script. This subdocument looks up the DepartmentName field data that the database stores in a separate table. In addition to adding a subdocument, the script sorts the input message by the LastName and FirstName elements. The following XML demonstrates what the input message would look like if a search were conducted on the CityName of Burbank (CityID=1).

```
<_doc>
<_form>e_employeelookup_employee_emplist.jsp</_form>
<_start>0</_start>
<_return>;employeedetail;CityName;OfficePhone;DepartmentName;
FirstName;LastName;Id;</_return>
<_count>10</_count>
```

```
<_ctxobj/>
<_ctxidfld/>
<_ctxidval/>
<CityID>1</CityID>
<search>1</search>
<_blankFields>;FirstName;false;LastName;false;DepartmentID;false
</_blankFields>
<__x>__y</__x>
<_callingform>e_employeelookup_search_search.jsp</_callingform>
<FirstName insertblank="false"/>
<LastName insertblank="false"/>
<DepartmentID insertblank="false"/>
</_doc>
```

Output

The script produces an XML document built from the schema and adapter specified in the sendDocQuery function. The XML output below is an example of the kind of data that could be returned using a similar script.

```
<recordset _count="10" _countFound="2" _more="0" _start="0">
  <employeedetail>
    <Id>10</Id>
    <FirstName/>
    <LastName>Burbank Agency</LastName>
    <OfficePhone>(408) 422-5501</OfficePhone>
    <CityName>Burbank</CityName>
    <DepartmentID>16</DepartmentID>
    <Department>
      <DepartmentName>Sales</DepartmentName>
    </Department>
  </employeedetail>
  <employeedetail>
    <Id>11</Id>
    <FirstName/>
    <LastName>Burbank Unit</LastName>
    <OfficePhone>(650) 572-9000</OfficePhone>
    <CityName>Burbank</CityName>
    <DepartmentID>19</DepartmentID>
    <Department>
      <DepartmentName>Technical Support</DepartmentName>
    </Department>
  </employeedetail>
<_form>e_employeelookup_employee_emplist.jsp</_form>
</recordset>
```

Description

This script displays the results list generated by the search form. The script uses two functions to change the data in the msg input message object. The first function checks the input message to determine the number of elements returned by the search results. If there any search results to return, the script

adds the Department subdocument to the msg message object. The second function sorts the input message by LastName and then FirstName. Using the adapter name and document schema name, this script then runs a SendDocQuery function to gather any search results that match those listed in the input message. The script then checks the <_countfound> tag generated by the query and determines if the return list is empty. If the list is empty, the script sets the msgReturn variable to the ListEmpty condition. This condition redirects users to the listempty form.

References

This section contains reference material to help you with scripting.

Sources for client-side JavaScript

- Devguru (JavaScript, VB script, HTML, etc.): http://www.devguru.com/
- HTML Writer's Guild: http://www.hwg.org/
- *JavaScript, The Definitive Guide*, David Flanagan, 3rd Edition, O'Reilly Publishing.
- JavaScript articles at IRT.org: http://www.tech.irt.org/articles/script.htm
- JavaScript Made Easy: http://www.easyjavascript.com/
- JavaScript Source: http://javascriptsource.com/
- JavaScript Source master list: http://javascript.internet.com/master-list/
- Netscape's Developer Site: http://developer.netscape.com
- Netscape's online JavaScript documentation: http://developer.netscape.com/docs/manuals/index.html?content=javascript.html
- Web Monkey: http://www.webmonkey.com/
- ZDNet JavaScript introduction: http://www.zdnet.com/devhead/filters/0,,2133214,00.html

JavaDocs for the main Archway package

For in-depth information about the Archway servlet and all the functions it supports, refer to the JavaDocs that are available on the Get-Resources Tailoring Kit installation CD. The JavaDocs are located in the \documentation\javadocs folder of your Get-Resources Tailoring Kit installation CD. To view the docs, launch the index.html file from this folder.

6 Document Schema Definitions

CHAPTER

This chapter describes document schema definitions and explains how they map data between Get-Resources and the back-end database. In addition, this chapter discusses how to use schema extensions to add new physical mappings to existing schemas.

This chapter covers the following topics:

Understanding document schema definitions

A document schema definition (also called a schema) is an XML file that instructs the Archway Document Manager how to query back-end databases and generate XML documents containing the query response. Schemas are mapping tools that determine which XML tags used in dynamically created documents map to the table and field names in a given back-end database. These generated XML documents provide the data that Get-Resources displays and processes.

All schemas consist of two types of definitions:

- Base definitions—The schema entries that provide a logical mapping between the XML tags generated in a document query to the Get-Resources interface are collectively referred to as the schema base definitions. The Archway Document Manager uses the base definitions to generate XML tags based on the elements listed in the schema. The Archway Document Manager converts the `name` value listed in an `<attribute>` element into an XML tag of the same name.

- Derived definitions—The schema entries that provide a physical mapping between the XML tags generated in a document query to the table and field names in the back-end database are collectively referred to as the schema derived definitions. The Archway Document Manager queries the tables and field names listed in the schema and creates an XML document with the results of the query. The Archway Document Manager converts the `table` and `field` values listed in the `<document>` and `<attribute>` elements into a SQL query.

Note: The document schema definitions used by Peregrine Studio are not the same as the schemas being proposed and developed by the W3C.

The base and derived definitions each have their own list of legal elements and attributes. For more information on schema elements and attributes and how to use them, refer to *Schema elements and attributes* on page 110.

How to use schemas

You can use schemas to present and store data from your back-end database in the Get-Resources interface. The Archway Document Manager uses schemas to create XML documents when a form onload script requests data from a back-end database. Typically, a form component such as a table or input field displays the requested schema data, but a script may also use the schema data to update or insert records in the back-end database as well.

You can tailor schemas in two ways:

- Create schema extensions. A schema extension is a separate file listing only the changes you make to an existing schema's logical or physical mappings. For example, you could create a schema extension to provide updated physical mappings when you upgrade your back-end database. Creating schema extensions is the preferred method of tailoring schemas as your changes are stored in separate files that can be easily carried over during an upgrade. For more information about schema extensions, see your *Get-Resources Administration Guide*.

- Create new schemas. You can create your own schemas to provide all form components in your project access to the custom logical and physical mappings you create for Get-Resources. For example, you could create a new schema to query a collection of custom-created tables and fields that you have added to your back-end database. While you can create new schemas from any text editor without the Get-Resources Tailoring Kit, you will need Peregrine Studio to configure and test any server onload scripts and form components that use your custom-built schemas.

Important: Do not directly edit an existing schema as any changes you make to existing logical and physical mappings will be overwritten when you upgrade to a newer version of Get-Resources.

Schema elements and attributes

All schemas use a standard set of XML elements and attributes that the Archway Document Manager recognizes. The following sections describe the XML elements and associated attributes that you can use to create valid schemas.

<?xml>

The <?xml> element is the standard XML namespace identifier. This element should always include the version attribute. All schemas require that this be the first element listed.

<schema>

The <schema> element is a required element of all schemas. The <schema> element functions as a container for the logical and physical mappings. The <schema> element does not have any attributes.

<documents>

Two sets of <documents> elements are required for each schema. One set of <documents> elements is the container for the logical mappings and the other set of <documents> elements is the container for the physical mappings.

Use in logical mapping

All schemas require one <documents> element where the name attribute has the value name="base". When this element has this name value, it becomes the container for the logical mappings.

Required attributes

- name. This attribute identifies the <documents> element container used by the logical mappings. This attribute must have the value name="base".

Optional attributes

■ *None.* There are no optional attributes for the logical mapping portion of the schema.

```
<?xml version="1.0"?>
<schema>

<documents name="base">
   ...
</documents>

...
```

Logical mappings always use name="base"

Use in physical mapping

All schemas require at least one <documents> element where the name attribute has the value of an adapter name such as name="ac". You can add one <documents> element for each adapter you want to provide physical mappings for. You can also support multiple versions of the same adapter if you use the version attribute.

Required attributes

■ name. This attribute determines what adapter the schema uses to make connections to the back-end database. The value of this attribute must be an adapter name such as name="ac".

Optional attributes

■ version. This attribute determines what version of the back-end database is required to use the physical mappings defined in this container. The value of this attribute must be a number recognized by the adapter.

```
<?xml version="1.0"?>
<schema>

...

<documents name="ac" version="3">
   ...
</documents>

<documents name="ac" version="4">
   ...
</documents>

...
```

You can add a <documents> element for each adapter

Each <documents> element can describe a different version

The Archway Document Manager uses the following rules to match the back-end database to the version listed in this attribute:

- If the <documents> element has *no* version attribute, then the Archway Document Manager accepts the physical mappings in this element if it cannot find another matching value.

- If the <documents> element has a version attribute value *greater* than the version number of the back-end database, then the Archway Document Manager ignores the physical mappings in this element.

- If the <documents> element has a version attribute value *less* than the version number of the back-end database, then the Archway Document Manager accepts the physical mappings in this element if it cannot find a higher matching value.

- If the <documents> element has a version attribute value *equal* to the version number of the back-end database, then the Archway Document Manager accepts the physical mappings in this element.

<document>

You must add at least two sets of <document> elements to create a valid schema – one set for the logical mappings and another set for the physical mappings. You can add additional <document> elements in the physical mapping section if you want to support multiple adapters or multiple versions of the same back-end database.

Use in logical mapping

The logical mapping section uses the <document> elements as a container for the XML document that the Archway Document Manager produces. All XML elements produced by this schema will be child elements of the <document> element.

Required attributes

- name. This attribute determines what XML element the Archway Document Manager generates as the top-level element in any generated document using this schema. The value of this attribute must match the file name of the schema (*without* the .xml extension).

Optional attributes

- ACLcreate. This attribute determines the default access control list for DocExplorer forms that use this schema. The value of this attribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see a **Create** button in DocExplorer forms that use this schema.

- ACLdelete. This attribute determines the default access control list for DocExplorer forms that use this schema. The value of this atrribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see a **Delete** button in DocExplorer forms that use this schema.

- ACLupdate. This attribute determines the default access control list for DocExplorer forms that use this schema. The value of this atrribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will be able to edit fields in DocExplorer detail forms that use this schema.

- create. This attribute determines if a subdocument using this element is visible in DocExplorer *create* forms. The value of this attribute must be either true or false. Set the value to create="true" if you want this subdocument to be available on DocExplorer create forms. Set the value to create="false" if you want to prevent this subdocument from being available on DocExplorer create forms.

- detail. This attribute determines if a subdocument using this element is visible in DocExplorer *detail* forms. The value of this attribute must be either true or false. Set the value to detail="true" if you want this subdocument to be available on DocExplorer detail forms. Set the value to detail="false" if you want to prevent this subdocument from being available on DocExplorer detail forms.

- docname. This attribute defines the external schema that you want the Archway Document Manager to use to create a subdocument. The value of this attribute must match the file name of the schema (*without* the .xml extension) that you want to use for the subdocument. You only need this attribute if you want to create a subdocument using an another schema.

- label. This attribute determines what name the schema has in DocExplorer forms that use this schema. The value of this attribute can be any text string. Typically, you will want to set this value to a user-friendly name describing the content of the schema.

- list. This attribute determines if a subdocument using this element is visible in DocExplorer *list* forms. The value of this attribute must be either true or false. Set the value to list="true" if you want this subdocument to be available on DocExplorer list forms. Set the value to search="false" if you want to prevent this subdocument from being available on DocExplorer list forms.

- **loadscript.** This attribute determines what ECMAScript runs when this schema is used in a DocExplorer form. The value of this attribute must be the Peregrine Studio name of the ECMAScript you want to run. You can use this script to load additional data for use by DocExplorer forms. This script uses the same XML message input as the form onload script. See *Document Schema Extensions* for examples of loadscripts.

- **preexplorer.** This attribute determines what ECMAScript runs when this schema is used in a DocExplorer form. The value of this attribute must be the Peregrine Studio name of the ECMAScript you want to run. You can use this script to make formatting changes to the XML message rendered by DocExplorer forms. See your Get-Resources deployment for examples of pre-explorer scripts. Pre-explorer scripts are located at the following path:
  <application server>\oaa\WEB-INF\apps*<package>*\jscript\preexplorer

- **search.** This attribute determines if a subdocument using this element is visible in DocExplorer *search* forms. The value of this attribute must be either true or false. Set the value to search="true" if you want this subdocument to be available on DocExplorer search forms. Set the value to search="false" if you want to prevent this subdocument from being available on DocExplorer search forms.

- **subtypeprop.** This attribute determines whether this element inherits the attribute properties of the parent <collection> element. The value of this attribute must be inherit if you use the attribute at all. If you want this element to inherit the attribute properties set the value to subtypeprop="inherit". If you want to specify the attribute properties for this element, do not include a subtypeprop attribute.

Use in physical mapping

The physical mapping section uses the <document> elements to define the SQL name of the back-end database table.

Required attributes

- **name.** This attribute determines what XML element the Archway Document Manager matches to a back-end database table. The value of this attribute must match the file name of the schema (*without* the .xml extension).

- **table.** This attribute identifies the table in the back-end database that the schema uses.The value of this attribute must be the SQL name of the table you want to use for source data. Each <document> element can only have one table attribute. To use data from other tables, you can create subdocuments within your schema.

Optional attributes

- attachtable. This attribute identifies the ServiceCenter table where references to attachments are located. The value of this attribute must be the SQL name of SerivceCenter table you want to use.

Note: You can only use this attribute when you are using ServiceCenter as your back-end database.

- field. This attribute identifies the field in the back-end database that you want the schema to use for document queries. The value of this attribute must be the SQL name of the field you want to use for the data source. You only need this attribute if you want to create a subdocument within your schema. You can also set this attribute to _null if there is no physical mapping for this document in your back-end database.

- insert. This attribute identifies the event name to be sent to ServiceCenter when Get-Services inserts (creates) a new record. The value of this attribute must be the SQL name of the ServiceCenter event.

Note: You can only use this attribute when you are using ServiceCenter as your back-end database.

- joinfield. This attribute identifies the field in the back-end database that you want the schema to use to query for additional information in another schema or table. The value of this attribute must be the SQL name of the field you want to use for the source data. You only need this attribute if you want to create a subdocument within your schema. The joinfield attribute defines what field will be the selection criteria in a SQL WHERE clause. The SQL equivalent of the joinfield is:

  SELECT <field> FROM <external table> WHERE *<joinfield>*=<joinvalue>

  If you do not provide a joinfield value, then the Archway Document Manager uses the field listed for the <attribute name="Id"> element as the joinfield.

- joinvalue. This attribute identifies the <attribute> element that has the value you want to use to query for additional information in another schema or table. The value of this attribute must be the name of an <attribute> element in the current schema. You only need this attribute if you want to create a subdocument within your schema. The joinvalue attribute defines what value a field must have in a SQL WHERE clause. The SQL equivalent of the joinvalue is:

  SELECT <field> FROM <external table> WHERE <joinfield>=*<joinvalue>*

If you do not provide a joinvalue value, then the Archway Document Manager uses the value returned for the <attribute name="Id"> element as the joinvalue.

- link. This attribute identifies the field in the back-end database that you want the schema to use to query for additional information in a table with lookup or link fields. The value of this attribute must be the SQL name of the field you want to use for the source data. You only need this attribute if you want to create a subdocument within your schema. In most cases, the link attribute is the same as the joinfield attribute. This value will only be different if the SQL name of the link field in the source table is different from the SQL name from the target field in the target table.

- preprocess. This attribute determines what ECMAScript runs *before* the Archway Document Manager connects to the back-end database. The value of this attribute must be the Peregrine Studio name of the ECMAScript you want to run. You can use this script to format the request sent to the back-end database. For example, you can add additional SQL commands or validate that all required fields are listed in the request. See your Get-Resources deployment for examples of pre-process scripts. Pre-process scripts are located at the following path:
  <*application server*>\oaa\WEB-INF\apps\<*package*>\jscript\schema

- postprocess. This attribute determines what ECMAScript runs *after* the Archway Document Manager receives a response from the back-end database. The value of this attribute must be the Peregrine Studio name of the ECMAScript you want to run. You can use this script to format the response sent from the back-end database. For example, you can sort the data by a particular criteria or return an error message if no records are found. See your Get-Resources deployment for examples of post-process scripts. Post-process scripts are located at the following path:
  <*application server*>\oaa\WEB-INF\apps\<*package*>\jscript\schema

- update. This attribute identifies the event name to be sent to ServiceCenter when Get-Resources updates an existing record. The value of this attribute must be the SQL name of the ServiceCenter event.

Note: You can only use this attribute when you are using ServiceCenter as your back-end database.

<attribute>

You must add at least two sets of <attribute> elements to create a valid schema – one set for the logical mappings and another set for the physical mappings.

Use in logical mapping

The logical mapping sections use the `<attribute>` elements to create an XML element in any document message built from this schema.

Required Attributes

- `name`. This attributes determines the XML tag that the Archway Document Manager generates when it uses the schema. The value of this attribute can be any string value. For example, if you set the value to `name="contact"` then the Archway Document Manager creates a `<contact>` XML tag. You must define at least one `<attribute>` element where the `name` attribute has the value `name="Id"`. This `<attribute>` element is required to uniquely identify each record returned by a schema query.

- `type`. This attribute determines what data format the elements uses as well as how Get-Resources renders the data in the user interface. The value of this attribute must be one of the following strings:

  - `attachment`—This element is a path and file name to an attachment. Get-Resources renders this element as a collection of attachment controls.

  - `boolean`—This element is a true or false string. Get-Resources renders this element as a check box.

  - `date`—This element is a date listing. Get-Resources renders this element as a date edit control that includes a popup calendar.

  - `datetime`—This element is a combined date and time listing. Get-Resources renders this element as a time edit control.

  - `id`—This element is a number that uniquely describes a back-end database record. Get-Resources renders this element as a single-line edit field.

  - `image`—This element is an image. Get-Resources renders this element as an imagefield.

  - `link`—This element is a subdocument described elsewhere in the schema. Get-Resources renders this element as a lookup field.

  - `memo`—This element is a text string. Get-Resources renders this element as a multi-line edit box.

  - `money`—This element is a currency amount. Get-Resources renders this element as a money field that includes a currency selection tool.

  - `number`—This element is an integer. Get-Resources renders this element as an editfield with spinner buttons.

- preload—This element is an executable script. Get-Resources runs the script listed in this element.

- string—This element is text. Get-Resources renders this element as an editfield.

- time—This element is a time listing. Get-Resources renders this element as a time edit control.

- url—This element is a Web site address. Get-Resources renders this element as an HREF link icon.

Note: The Archway Document Manager does not validate that the contents of an element matches the type attribute listed for it.

Optional attributes

- access. This attribute determines whether the field described by this element accepts updates or inserts in the back-end database or whether it is a read-only field. The value of this attribute must be either r or null. Set the value to access="r" if you want to make this element read-only. Clear the value or remove the attribute if you want to enable updates and inserts to this field.

- ACLcreate. This attribute determines the default access control list for DocExplorer forms that use this element. The value of this atrribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see this element in DocExplorer *create* forms that use this schema.

- ACLdetail. This attribute determines the default access control list for DocExplorer forms that use this element. The value of this atrribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see this element in DocExplorer *detail* forms that use this schema.

- ACLlist. This attribute determines the default access control list for DocExplorer forms that use this element. The value of this atrribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see this element in DocExplorer *list* forms that use this schema.

- ACLsearch. This attribute determines the default access control list for DocExplorer forms that use this element. The value of this atrribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see this element in DocExplorer *search* forms that use this schema.

- `create`. This attribute determines if the element is visible in DocExplorer *create* forms. The value of this attribute must be either true or false. Set the value to `create="true"` if you want this field to be available on DocExplorer create forms. Set the value to `create="false"` if you want to prevent this field from being available on DocExplorer create forms.

- `detail`. This attribute determines if the element is visible in DocExplorer *detail* forms. The value of this attribute must be either true or false. Set the value to `detail="true"` if you want this field to be available on DocExplorer detail forms. Set the value to `detail="false"` if you want to prevent this field from being available on DocExplorer detail forms.

- `label`. This attribute determines what name the element has in DocExplorer Available Field list. The value of this attribute can be any text string. Typically, you will want to set this value to a user-friendly name describing the content of the field.

- `list`. This attribute determines if the element is visible in DocExplorer list forms. The value of this attribute must be either true or false. Set the value to `list="true"` if you want this field to be available on DocExplorer list forms. Set the value to `search="false"` if you want to prevent this field from being available on DocExplorer list forms.

- `required`. This attribute determines if this element requires a value in order to insert or update a record in the back-end database. The value of this attribute must be either true or false. Set the value to `required="true"` if you want to make the element a required input field when it is added to DocExplorer forms.

- `search`. This attribute determines if the element is visible in DocExplorer *search* forms. The value of this attribute must be either true or false. Set the value to `search="true"` if you want this field to be available on DocExplorer search forms. Set the value to `search="false"` if you want to prevent this field from being available on DocExplorer search forms.

Use in physical mapping

The physical mapping sections use the `<attribute>` elements to define the fields in the back-end database that map to each logical mapping.

Required Attributes

- `name`. This attributes determines the XML tag in which the Archway Document Manager places query results. The value of this attribute must match an element defined in the logical mapping section.

- **field.** This attribute identifies the field in the back-end database that you want the schema to use for document queries. The value of this attribute must be the SQL name of the field you want to use for the data source. You can also set this attribute to _null if there is no physical mapping for this field in your back-end database.

Optional attributes

- **link.** This attribute identifies a lookup or link value to another table. The value of this attribute must be the SQL name of the link. You will only need this attribute if you want to query information from a field in one table that links to another field in a linked table. The link attribute defines what field is the selection criteria in a SQL WHERE clause. The SQL equivalent of the link is:

  SELECT <linkfield> FROM <linktable> WHERE *<link>*=<field>

- **linkfield.** This attribute identifies the target field called by a lookup or link value to another table. The value of this attribute must be the SQL name of the target field. You will only need this attribute if you want to query information from a field in one table that links to another field in a linked table. The linkfield attribute defines what field is selected. The SQL equivalent of the link is:

  SELECT *<linkfield>* FROM <linktable> WHERE <link>=<field>

- **linkkey.** This attribute identifies the field, lookup, or link that connects two fields in linked tables. The value of this attribute must be the SQL name of the linking field. You will only need this attribute if you want to query information from a field in one table that links to another field in a linked table. The linkkey attribute defines what field is selected. The SQL equivalent of the link is:

  SELECT <linkfield> FROM <linktable> WHERE *<linkkey>*=<field>

  If you do not define a linkkey value, then the Archway Document Manager uses the link attribute as the linkkey.

- **linktable.** This attribute identifies the target table called by a lookup or link value. The value of this attribute must be the SQL name of the target table. You will only need this attribute if you want to query information from a field in one table that links to another field in a linked table. The linktable attribute defines what table is named in a SQL FROM clause. The SQL equivalent of the linktable is:

  SELECT <linkfield> FROM *<linktable>* WHERE <link>=<field>

- linktype. This attribute defines how the Archway Document Manager performs document inserts and updates. The value of this attribute must be either soft or hard:

- soft—The Archway Document Manager queries the back-end database using the locations listed in the `linktable` and `linkfield` attributes, and sets the `link` attribute to the value to the query result.

- hard—The Archway Document Manager creates a new record in the back-end database at the location listed in the `linktable` and `linkfield` attributes. The Archway Document Manager retrieves the `linkkey` value for the new record and saves it in the field listed in the `link` attribute.

If you do not specify a `linktype` value, then it defaults to `soft`. You will only need this attribute if you want to query information from a field in one table that links to another field in a linked table.

\<collection\>

This is an optional element that you can use to create subdocuments where more than one item can be returned for the document you query. For example, you can create a set of \<collection\> elements to query for all the requests that a particular user has open. In database terminology, a \<collection\> element returns the records from an intersection table. You must add one set of \<collection\> elements for each multiple item subdocument you want to create.

Use in logical mapping

The logical mapping section uses the \<collection\> elements to create the XML elements that the subdocuments use.

Required attributes

- `name`. This attribute determines what XML element the Archway Document Manager generates as the top-level element in any generated document using this schema. The value of this attribute must match the file name of the schema (*without* the .xml extension) that the subdocument uses.

Optional attributes

- `ACLcreate`. This attribute determines the default access control list for DocExplorer forms that use this subdocument. The value of this attribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see a **Create** button in DocExplorer forms that use this schema.

- ACLdelete. This attribute determines the default access control list for DocExplorer forms that use this subdocument. The value of this atrribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see a **Delete** button in DocExplorer forms that use this schema.

- ACLupdate. This attribute determines the default access control list for DocExplorer forms that use this subdocument. The value of this atrribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will be able to edit fields in DocExplorer detail forms that use this schema.

- create. This attribute determines if a subdocument using this element is visible in DocExplorer *create* forms. The value of this attribute must be either true or false. Set the value to create="true" if you want this subdocument to be available on DocExplorer create forms. Set the value to create="false" if you want to prevent this subdocument from being available on DocExplorer create forms.

- detail. This attribute determines if a subdocument using this element is visible in DocExplorer *detail* forms. The value of this attribute must be either true or false. Set the value to detail="true" if you want this subdocument to be available on DocExplorer detail forms. Set the value to detail="false" if you want to prevent this subdocument from being available on DocExplorer detail forms.

- label. This attribute determines what name the subdocument has in DocExplorer forms that use this schema. The value of this attribute can be any text string. Typically, you will want to set this value to a user-friendly name describing the content of the schema.

- list. This attribute determines if a subdocument using this element is visible in DocExplorer list forms. The value of this attribute must be either true or false. Set the value to list="true" if you want this subdocument to be available on DocExplorer list forms. Set the value to search="false" if you want to prevent this subdocument from being available on DocExplorer list forms.

- search. This attribute determines if a subdocument using this element is visible in DocExplorer *search* forms. The value of this attribute must be either true or false. Set the value to search="true" if you want this subdocument to be available on DocExplorer search forms. Set the value to search="false" if you want to prevent this subdocument from being available on DocExplorer search forms.

Use in physical mapping

The physical mapping section uses the `<collection>` elements to define the SQL name of the back-end database table.

Required attributes

- `name`. This attribute determines what XML element the Archway Document Manager matches to a back-end database table. The value of this attribute must match the file name of the schema (*without* the .xml extension).

Optional attributes

- *None*. There are no optional attributes for the physical mapping portion of a `<collection>` element.

Documents

The Archway Document Manager uses schemas to create documents, which are XML messages created from the following components:

- Schema logical definitions. The schema logical definitions determine what XML elements make up the generated document.

- The return values of database queries. The Archway Document Manager uses the schema physical mappings to create database queries. The return values of these queries determine the content of the elements and attributes of the generated document.

- ECMAScript formatting. ECMAScripts can modify a document before and after any queries have been made to the back-end database.

The final output of these three processes is an XML document that the Archway Document Manager renders as HTML in the Get-Resources interface.

You can see the raw Get-Resources XML documents by enabling the **Show form information** option from the Administration settings. The form information window displays the following document information:

- **Script Input**. This tab displays the document submitted to the current form from the output of a previous form. For example, a list form displays the output of a prior search form. This document is passed to the form onload script as an input parameter.

- **Script Output**. This tab displays the document generated by the output of the current form's onload script. Typically, each onload script invokes a schema that queries the back-end database for relevant information. For example, a service form will invoke a database query through the incident schema.

- **PreXSL.** This tab displays the document after the Archway servlet has processed the document and prepared it to be rendered by the client-side browser.

Subdocuments

Each Get-Resources form typically maps to one schema, which in turn maps to one table in the back-end database. In order to collect and represent data from multiple schema and database sources, you must create subdocuments.

Subdocuments are XML messages added to the current document that query additional schemas and tables. You can create subdocuments in one of two ways:

- You can add a new <document> element inside an existing <document> element if the result of the query will be *one and only one* subdocument.
- You can add a <collection> element inside an existing <document> element if the result of the query will be a collection of *one or more* subdocuments.

The following sections examples of each method.

Creating subdocuments with the <Document> element

Each <document> element is intended to return one subdocument, that is, one record set. For example, you can create sudocument to query for the contact name for a specific request, but each request should only have one contact name.

Schema

The following schema segment illustrates how to add a subdocument using the <document> element.

Logical mapping for
subdocument – EndUser ——

```
<documents name="base">
  <document name="Request" label="Request"...>
    <attribute name="Id" type="id".../>
    <attribute name="Number" type="string" label="Number".../>
    <attribute name="Purpose" type="string" label="Purpose".../>
    ...
    <document name="EndUser" docname="Employee" label="End User"/>
    ...
  <document>
<documents>
```

Physical mapping for
subdocument – EndUser ——

```
<documents name="ac" version="4">
  <document name="Request" table="amRequest"...>
    <attribute name="Id" field="lReqId"/>
    <attribute name="Number" field="ReqNumber"/>
    <attribute name="Purpose" field="ReqPurpose"/>
    ...
    <document name="EndUser" docname="Employee" table="amEmplDept"
      field="lUserId" link="lUserId" joinfield="lEmplDeptId"
      joinvalue="EndUserId"/>
    ...
  <document>
<documents>
```

XML Output

The Archway Document Manager produces an XML document with the following structure. You can view such documents from the Script Input and Script Output tabs of the Form Information window. The values stored in the XML elements vary depending on the actual user record you select.

Elements from schema
mapping – Id, AssetTag ——

Joinvalue – EndUserId ——

```
<Request>
  <Id>32097</Id>
  <Number>REQ000042</Number>
  <Purpose>Purpose 1</Purpose>
  ...
  <EndUserId>15630</EndUserId>
  ...
</Request>
```

Creating subdocuments with the <Collection> element

Each <collection> element is intended to return more than one subdocument or record set. For example, you can create a query to return all the requests belonging to a particular contact.

Schema

The following schema segment illustrates how to add a subdocument using the <collection> element.

Logical mapping for
subdocuments –
RequestLine

```
<documents name="base">
  <document name="Request" label="Request"...>
    <attribute name="Id" type="id".../>
    <attribute name="Number" type="string" label="Number".../>
    <attribute name="Purpose" type="string" label="Purpose".../>
    ...
    <collection    name="RequestLines" label="Composition">
         <document    name="RequestLine"/>
    </collection>
    ...
  <document>
<documents>
```

No physical mapping for
subdocuments –
RequestLine. Therefore,
physical mapping
defaults to that listed in
RequestLine schema

```
<documents name="ac" version="4">
  <document name="Request" table="amRequest"...>
    <attribute name="Id" field="lReqId"/>
    <attribute name="Number" field="ReqNumber"/>
    <attribute name="Purpose" field="ReqPurpose"/>
    ...
    <!-- No physical mapping for the RequestLines collection. -->
    ...
  <document>
<documents>
```

Logical mapping for
RequestLine schema

Logical mapping for
subdocuments –
RequestLine

```
<documents name="base">
<document name="RequestLine" label="Request Line"...>
    <attribute name="Id" type="id" search="false" list="false"
     detail="false" create="false" />
    ...
    <collection name="RequestLines" label="Composition" detail="true"
     create="true">
       <document name="RequestLine" table="_null"/>
    </collection>
    ...
  <document>
<documents>
```

Physical mapping for
subdocuments –
RequestLines

```
<documents name="ac" version="4.0">
  <document name="RequestLine" table="amReqLine"...>
    <attribute name="Id" field="lReqLineId" />
    ...
    <collection    name="RequestLines" label="Composition">
         <document    name="RequestLine" table="_null"
         joinfield="lParentId" />
    </collection>
    ...
  <document>
<documents>
```

XML Output

The Archway Document Manager produces an XML document with the following structure. You can view such documents from the Script Input and Script Output tabs of the Form Information window. The values stored in the XML elements vary depending on the actual user record you select.

Elements from schema mapping – Id, AssetTag

Subdocuments – RequestLine

```
<Request>
  <Id>32098</Id>
  <Number>REQ000043</Number>
  <Purpose>Purpose 2</Purpose>
  ...
  <RequestLines _count="-1" _countFound="3" _more="0" _start="0">
    <RequestLine>
      <Id>32100</Id>
      <RequestId>32098</RequestId>
      <RequestNumber>REQ000043</RequestNumber>
      ...
    </RequestLine>

    <RequestLine>
      <Id>32101</Id>
      <RequestId>32098</RequestId>
      <RequestNumber>REQ000043</RequestNumber>
      ...
    </RequestLine>

    <RequestLine>
      <Id>32102</Id>
      <RequestId>32098</RequestId>
      <RequestNumber>REQ000043</RequestNumber>
      ...
    </RequestLine>
  </RequestLines>
</Request>
```

III Tailoring Procedures and Testing

SECTION

This section lists and describes all the tailoring and testing procedures necessary to tailoring your Get-Resources project.

This sections includes:

7 Tailoring Tasks

CHAPTER

The following chapter lists all the tailoring tasks you can perform with the Get-Resources tailoring kit.

This chapter covers the following topics:

Tailoring workflow

You can use this flowchart to determine how to tailor Get-Resources.

Sample Tialoring Tasks

Enable User Self-Registration
Enable Change Password
Enable Automatic Login
Enable Integrated Windows Authenication
Enable/Disable Personalization
Assign Global Capability Words
Set a maximum row count on tables

Change theme
Add/Remove content from portal page
Perform and save document searches
Change time zone

Change the images used in a theme
Change the size or number of frames
Change the layout of frames
Change how form components are rendered (XSL)
Change the style sheet

Add a field on a form
Remove a field from a form
Make a field read-only or required
Change a field's label
Change a form's title or instructions
Set permissions to update, create, delete documents on a form.

Add or hide a field on the Available Fields column
Display or hide a field on a particular Personalization form
Change a field's attribute type (string, boolean, number, etc.)
Call a script in addition to the form onload script

Add custom forms to your project
Add form components to a form without Personalization
Change the schema used by a form component
Change the onload script launched by a form
Create a new schema

[Flowchart]

Start

Is there an Admin setting? —Yes→ Login to admin.jsp, change the setting, and reset the server.
No

Is a portal component available? —Yes→ Enable and personalize the portal component
No

Are you tailoring a theme? —Yes→ Create a custom theme and change the images and XSL files as needed
No

Is personalization available? —Yes→ Is there a personalization setting? —Yes→ Use Personalization to tailor the form
No (from Is there a personalization setting?) → Are you adding a field or script? —Yes→ Create a schema extension and then use Personalization to tailor the form
No (from Is personalization available?) and No (from Are you adding a field or script?) →

Use Peregrine Studio to tailor the form

Finish

List of tailoring tasks

The following sections list the tailoring tasks you can perform with the Get-Resources Tailoring Kit and Peregrine Studio.

Forms and form components

You can tailor forms and form components in the following ways:

DocExplorers

You can use Peregrine Studio to add and customize DocExplorers in the following ways:

Scripting

You can use the following scripting methods for tailoring:

Schemas

You can tailor schemas in the following ways:

Data validation

You can use Peregrine Studio to add data validation in the following ways:

Default values

You can use Peregrine Studio to assign default values to items in the following ways:

Translation

You can translate your tailored forms in the follow ways.

Tailoring forms and components

Each page displayed in Get-Resources consists of a form and several form components. Each form also has the following supporting elements:

- An onload script that gathers the data that the form displays or processes information from the previous form.
- A schema, which maps to fields in the database and determines what information to display.

For a complete list of each component available in Studio, see *Peregrine Studio Components* on page 53.

You can change a form's title, instructions, onload script, and component labels. You can also hide a form component and make a form read-only.

To tailor Get-Resources forms:

Step 1 Open the project file you want to tailor in Peregrine Studio.

Step 2 Select or create a package extension in which to save your changes.

Step 3 Open your browser and log in to Get-Resources.

Step 4 Navigate to the form you want to tailor by doing one of the following:

- Click the Studio address in the Form Information banner. Peregrine Studio will appear as the active window and display the current form's properties page.
- In Peregrine Studio, locate the form in the Project Explorer.

Step 5 Modify the Get-Resources form in Peregrine Studio.

Step 6 Save the project file.

Step 7 Rebuild the project file.

> **Tip:** If you have only made changes to one or more forms in an activity or module, use the Differential Build option ( ) to build just the components that have changed. This option will reduce the time needed to build your Peregrine Studio project.

Step 8 Restart your application server to clear the cache.

Step 9 Refresh the browser to reload the form you modified.

**Step 10** Review your changes and test the added functionality.

Tip: If you want to test new access right settings for your components, log on to Get-Resources with several different users with different access rights.

Changing a form's title

Each form displays a title at the top of the navigation menu. If you want to change or remove the title displayed for a particular form, set the following form properties.

To change a form title:

1 Open the form's properties in Peregrine Studio.

2 In the Title (en) field, enter the new form title

3 Click the check mark button ( ) at the right of the field to accept the new title.

4 Save and build your project file.
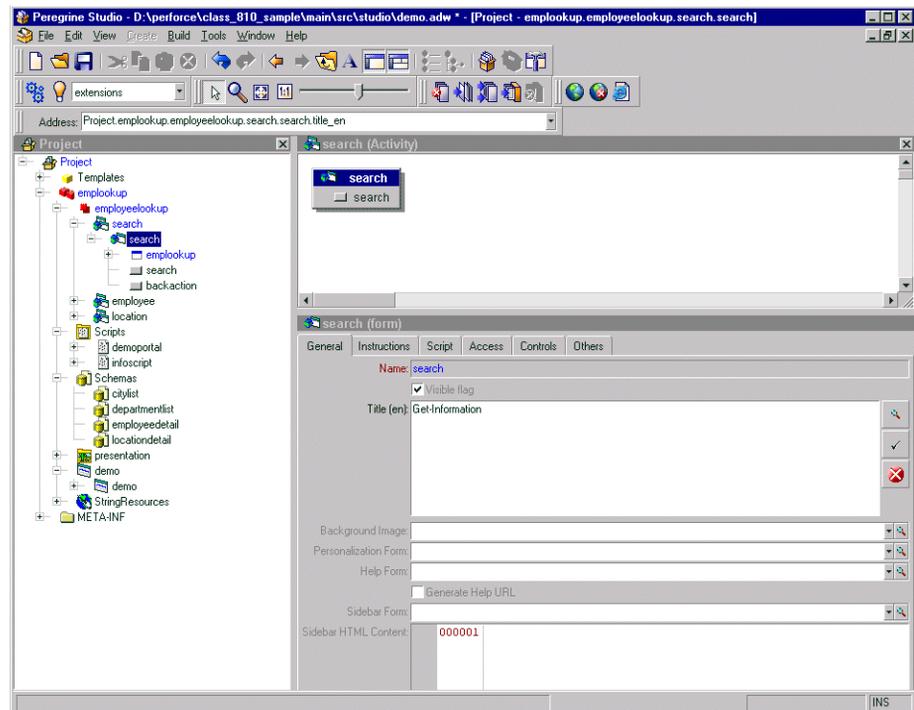
Changing a form's instructions

Most forms display a set of instructions at the top of the frame. You can change the instructions to match any changes you make to the form's interface.

To change form instructions:

1 Select the form in the Project Explorer.

2 Select the Instructions tab in the Properties window.

3 In the Instructions (en) field, enter the new form instructions.

4 Click the check mark button (![check]) at the right of the field to accept the new form instructions.

5 Save and build your project file.
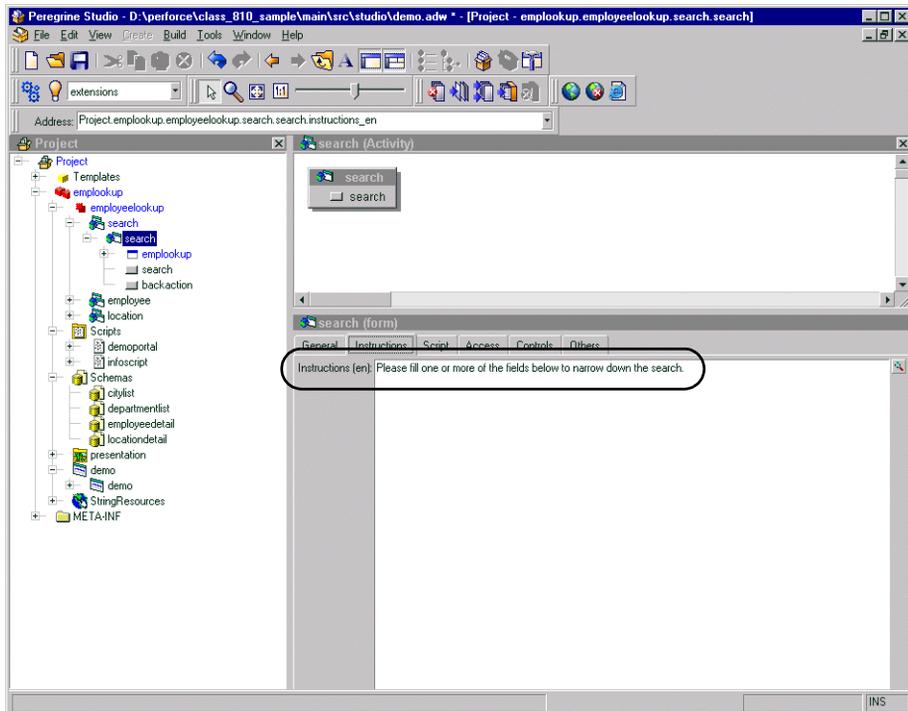
Changing a form's onload script

A form's onload script gathers all the data that the form displays, or processes information from the previous form. Many onload scripts also invoke schemas to present back-end database information in a format that is easier to map to particular form fields or form components.

To change the onload script invoked by a form:

1 Select the form in Studio.

2 Click the Script tab in the Properties window.

3 In the Server Onload Script field, enter or select the script you want to invoke when this form is loaded. You can use the drop-down list to select any of the scripts saved in your project file.

4 Save and build your project file.

5 Restart your application server.

Changing a form component's label

Many form components contain a label that is displayed next to or above the form component. Some of the most commonly configured form components are the field form components (check box, select box, edit field, and so forth).

To change a component label (field label):

1 Select the form in the Project Explorer.

2 On the General tab, select the Label (en) field, enter the new form component label, and press ENTER.

3 Save your project.

4 Build your project file.
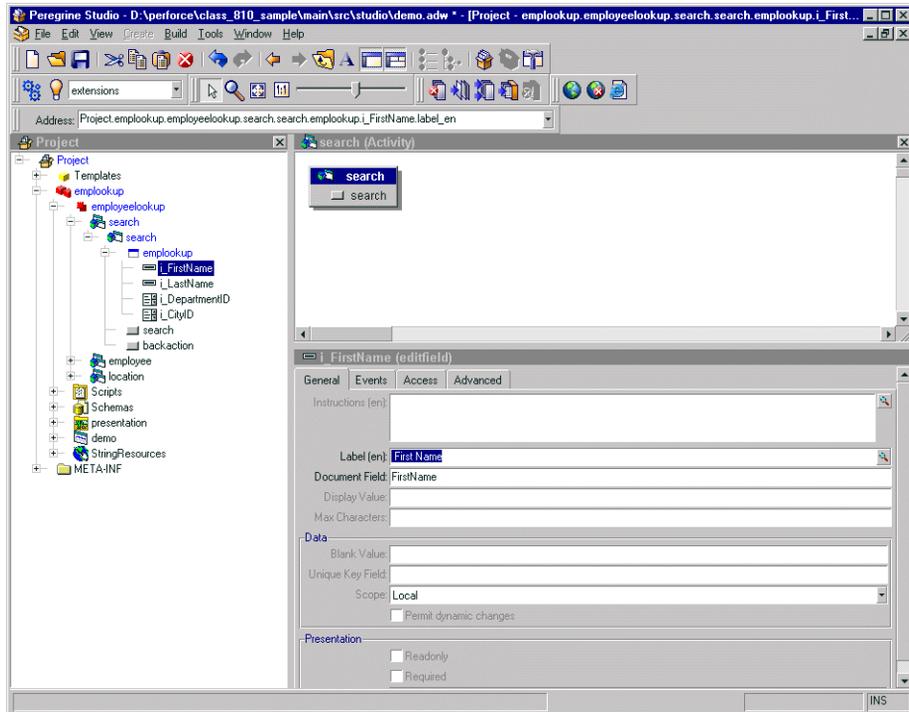
Hiding a form component

All form components have a Visible flag property that hides or displays the component in the Web application interface. If you want to remove a form component from the interface but still have it available in Peregrine Studio, you can toggle the form component's Visible flag to No. This prevents the form component from being part of the next Peregrine Studio build. Non-visible (and thus non-built) form components are displayed with a red X over the form component icon in the Project Explorer tree.

To hide a form component in the interface:

1 Select the form in the Project Explorer.

2 On the Advanced tab, clear the Visible flag option.

3 Save your project.

4 Build your project file.
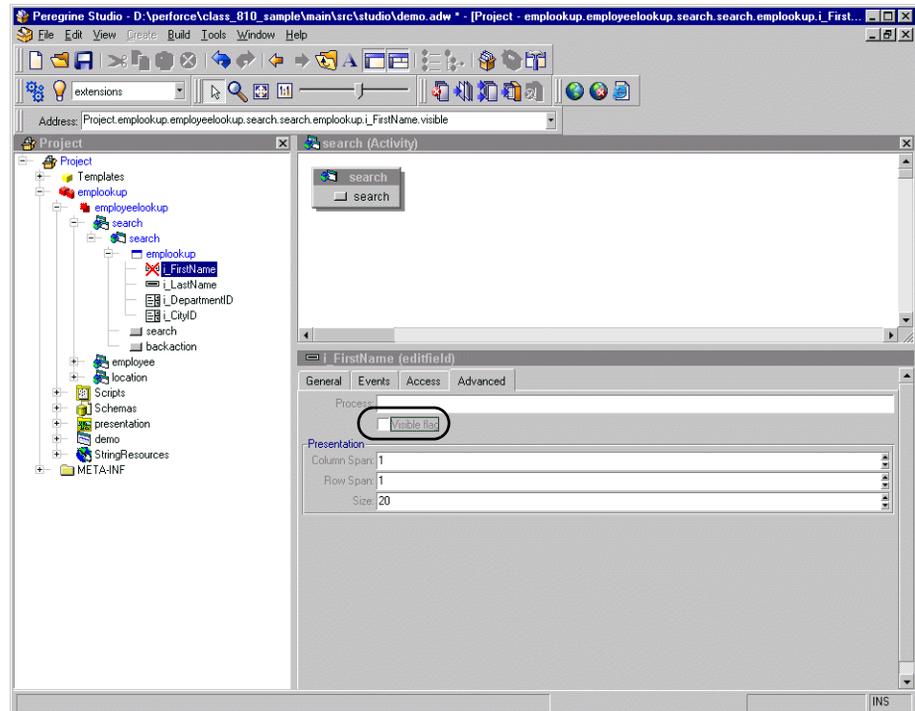

Changing a form component to read-only

Certain form components such as edit fields and text areas are available for users to enter and change information. If you want to restrict these form components so that they only display data, you can set the readonly attribute for the form component. The data displayed by a readonly form component will no longer have a bounding box or area to indicate that it can be edited or changed.

You can change a form component back to its original state by removing the readonly attribute.

To make a form component read-only:

1 Select the form in the Project Explorer.

2 On the General tab, select the Readonly check box.

3 Save your project.

4 Build your project file.
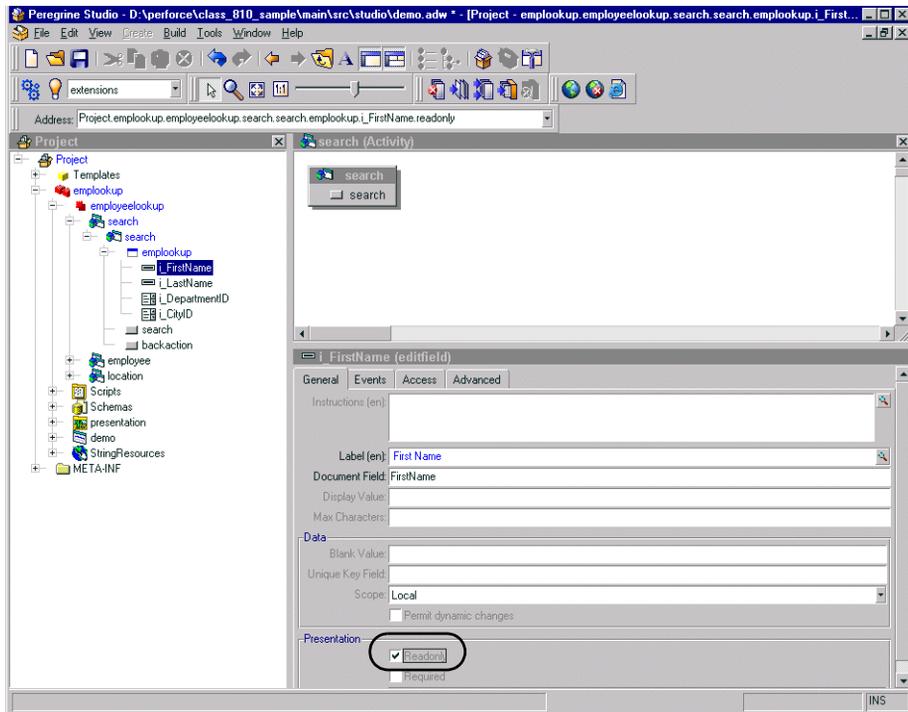
Changing the schema that a form component uses

Certain form components such as selectfields and simple tables use a schema to determine what information to display. You can change the information these form components display by changing the schema defining the document fields. In some cases you may also need to change other form component attributes that depend on the fields defined in the schema.

To change the schema that a form component uses:

1 Select the form in the Project Explorer.

2 Click the Advanced tab.

3 In the Databound section, select the **Document** field, and enter or select the name of the schema that you want to use as the source document for this form component.
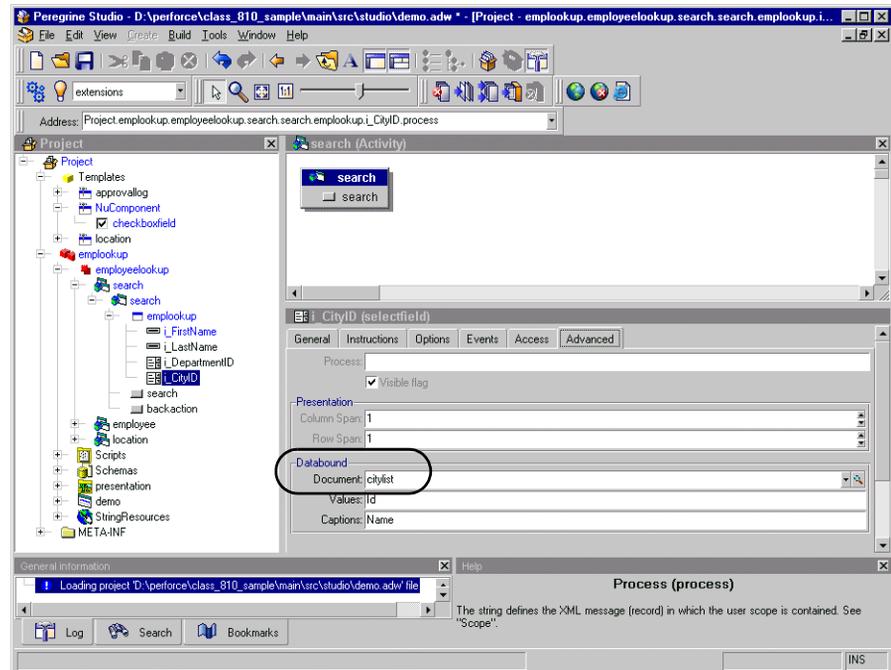
4 Save and build your project file.

Changing the document field that a form component uses

Certain form components such as selectfields and table columns use a particular document field of a schema to determine what information to display. You can change the information these form components display by changing the document fields these components use.

Note: The list of document fields available to a form component is determined by the schema used. Peregrine Studio does not validate the document field you select.

To change the document field that a form component uses:

1 Select the form component in Peregrine Studio to display the component's properties.

2 In the Document Field field, enter the name of the field in the XML message where this form component's information is stored.

Note: The field you select must be defined as an attribute in the schema defined in the form component's properties.

3 Save your project.

4 Build your project file.
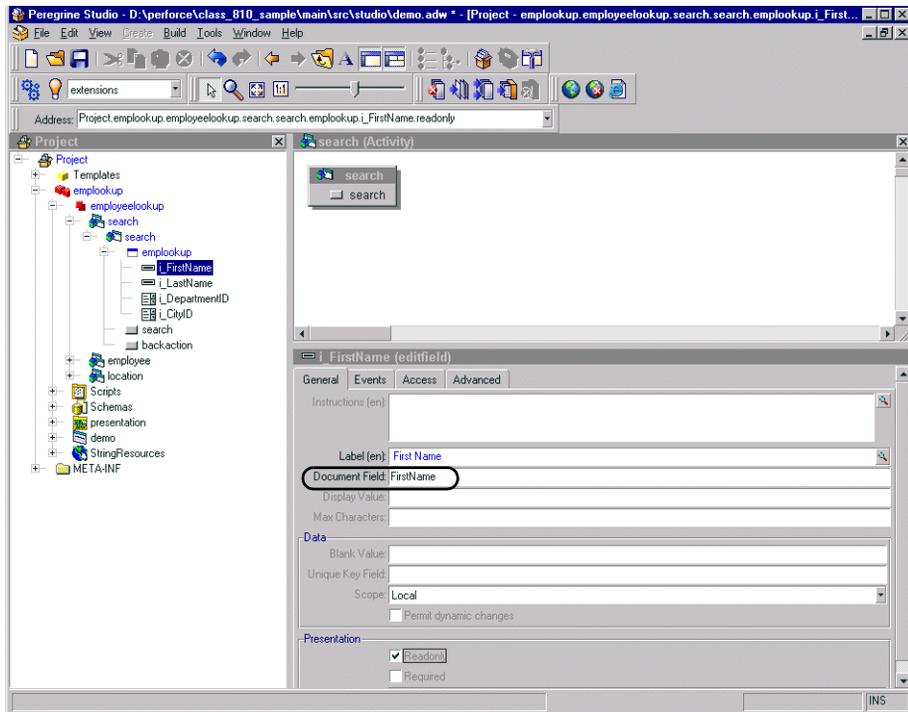
Format of document field name

The Document Field attribute of forms is always mapped to an element in the Message object returned by the form's onload script.

The Archway servlet formats Message objects as XML files using the tag definitions and back-end database table and field information that the schemas provide.

The Document Field attribute of a form component must map to an <attribute> element in a schema.

You can specify the Document Field attribute that a form component uses in one of several ways:

- If the Document Field attribute has a unique <attribute> name in the schema, you can list just the <attribute> name.

- If the Document Field attribute is repeated in the schema, you must specify the nested <document> name or names and the <attribute> name. The <document> name and the <attribute> name must be separated by a slash character (/).

- If the Document Field attribute is part of a nested <document> element, you have the choice of either listing the <attribute> name by itself or specifying the some or all of the path using the syntax of <documents>/<document>/<attribute>. This syntax allows Web application developers to specify as much or as little of the document path as is needed to create a field attribute mapping.

Example

Suppose you are creating a form where users can review and submit asset requests. A typical asset request may be formatted as the following XML message:

```
<request>
  <Number>012345</Number>
  <Purpose>Asset Management</Purpose>
  <EndUser>
    <FirstName>Michaela</FirstName>
    <LastName>Tossi</LastName>
  </EndUser>
  <Requester>
    <FirstName>Richard</FirstName>
    <LastName>Hartke</LastName>
  </Requester>
</request>
```

In this case, the <FirstName> and <LastName> tags are repeated in two different sections of the XML message. To display these tags in a form, you will need to specify more of the document path when you enter the path of the Document Field attribute. The entries below illustrate the minimum document path needed for the Document Field attribute in a form component.

```
Number
Purpose
EndUser/FirstName
EndUser/LastName
Requester/FirstName
```

```
Requester/LastName
```

You can also specify the Document Field attribute path using all the elements of the XML message. The following entries illustrate the full document path that can be used for the Document Field attribute in a form component.

```
request/Number
request/Purpose
request/EndUser/FirstName
request/EndUser/LastName
request/Requester/FirstName
request/Requester/LastName
```

The number of elements that you must specify in the document path is determined by how you set up your schemas.
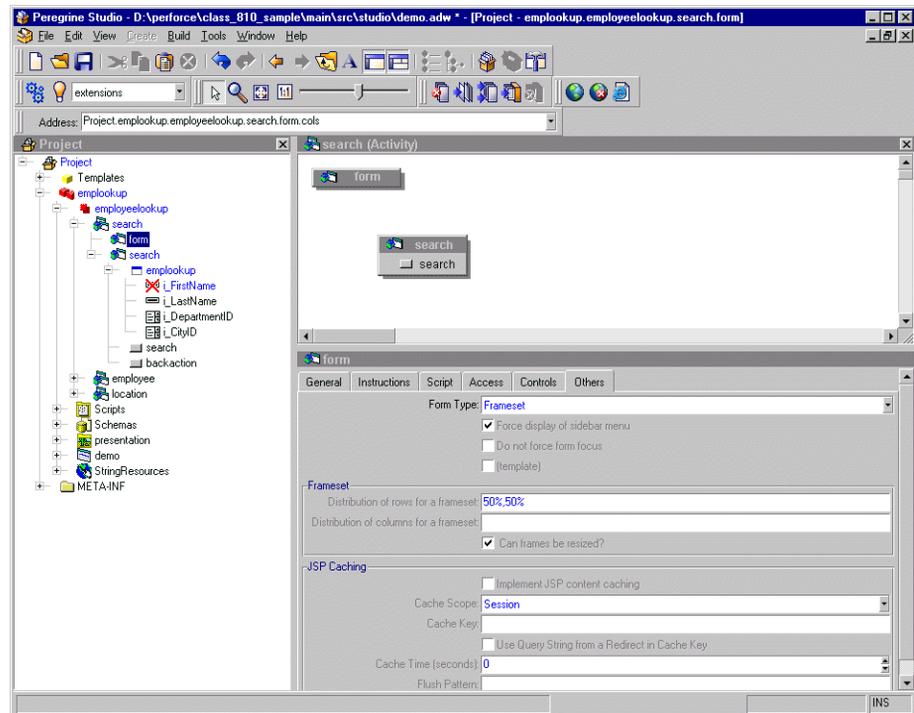
Displaying a form within a frameset

You can display forms within multiple frames by creating a special frameset form. All frames within a frameset form will be displayed within the frame normally reserved for forms.

To display forms within a frameset:

1 Right-click the activity where you want the frameset form to be, point to New, and then click Form.

2 Click the Others tab.

3 Select Frameset from the Formtype drop-down list box.

4 Enter row and column sizes in the Frameset pane.

Note: You can use percentage to describe frameset size properties.

5 Create a new form for each frame in the frameset form.

6 Create a redirection under the frameset form for each target form in the frameset.

7 Save your project.

8 Build your project file.

To display the form title within a frameset:

1 Open the frameset form's component properties in Studio.

2 Create a new server onload script within your project.

3 Add the following lines to the script:

```
top.setTitle("My Title Text");
```

Where *My Title Text* is the title you want to display at the top of the frameset.

4 Open the component properties page for the target form within the frameset.

5 Click the Script tab.

6 Select the server script you created in step 2.

7 Save your project.

8 Build your project file.

Adding Get-Resources to an existing frameset

You can add Get-Resources to an existing frameset to incorporate into your corporate intranet. To do this, you will need to edit a JavaScript file within your project file and add a reference to Get-Resources to the parent frameset.

To add Get-Resources to an existing frameset:

1 Open the following file in a text editor:

    <tomcat installation>\webapps\oaa\js\setDomain.js

or locate the file in the equivalent directory in your application server.

2 Add the following line to the bottom of the script:

    setDomain(server name);

where server name is the name of the server where the parent frameset is located.

3 Save the file.

4 Add the following line to each JSP file that will include Get-Resources in a frameset. These files must be saved on the server listed in step 2.

    <script language="JavaScript" SRC="js/setDomain.js">
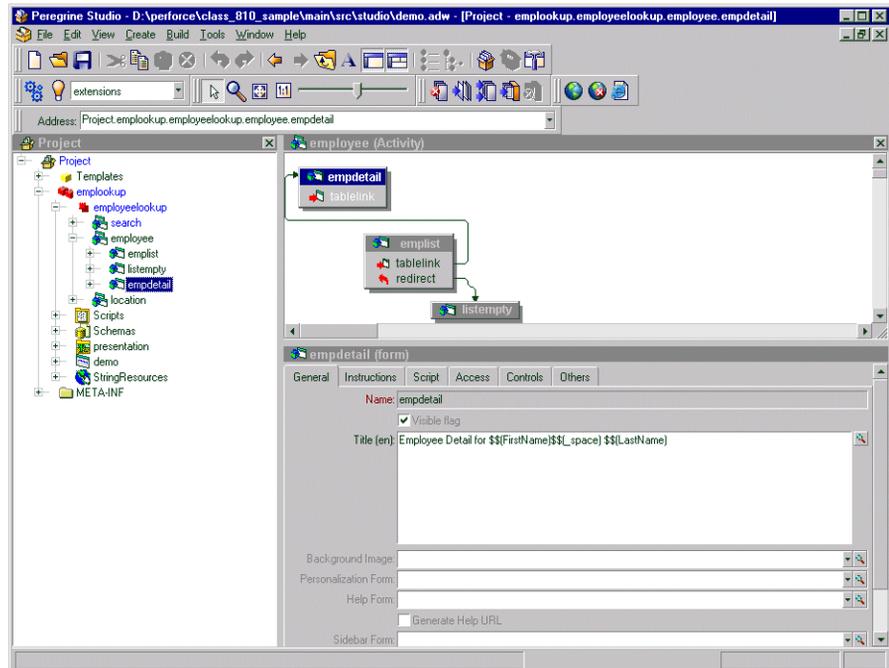   </script>

5 Save the updated JSP files.

Displaying a script variable in a form component

You can use script variables to reuse information gathered from other forms in form components such as form titles and instructions.

All script variables begin with a double dollar sign notation and then display the variable name in parentheses; for example, $$(FirstName). All variable names map to an XML element name in the script output of a form. Thus the script variables $$(FirstName) and $$(LastName) map to the elements <FirstName> and <LastName> in the XML output of a script.

The contents of each variable are displayed in the form title.

Note: You must select the **Display form information** option from **Administration** > **Settings** in order to see the Script Input and Script Output options.

Variable names can also include schema attribute names or nested elements names using a slash notation. For example, the buyer script uses the $$(Price/currency) variable to pass information from the currency attribute of the <Price> element. Using the sample data, the $$(Price/currency) variable would pass 1119.00 for the <Price> and USD for the currency attribute.

Creating a portal component

Portal components are special forms that display on the Peregrine Portal home page within special portal frames. To create your own portal components you need:

- Get-Resources packages and source code (included with the Get-Resources tailoring kit)
- Peregrine Studio

To create a portal component:

1 Open the Get-Resources project in Peregrine Studio.

2 Right-click the Group of Modules node to which you want to add a portal component and then select **New** > **Group of Portal Components**.

   You do not have to add another Group of Portal Components if one already exists in your project.

3 Right-click the Group of Portal Components node from the navigation tree and select **New** > **Portal Component**.

4 Enter the following properties for the portal component:

   a **Label (en)**. Enter the name you want the portal component to have in the Add/Remove content page.

   b **Column type**. Select either wide or narrow. This setting determines the size of the portal frame where Get-Resources displays the portal component.

   c **Height of IFRAME**. Enter a height value if you plan to display this portal component from WebSphere Portal Server.

5 Right-click on the new portal component and select **New** > **Contents**.

   A standard form page is added.

6 Enter any form components, onload scripts, parameters, or access restrictions you want the portal component to have.

   Tip: You can use existing Get-Resources portal components as a template.

   Keep in mind the following considerations:

   - Portal components have less space then normal forms to display information. You should design your form component to fit in either a narrow or wide portal component frame.

- Portal components cannot include the redirection form component. If you want to direct users to another form or HTML page, you will need to use the Business View Authoring tool.

- You can import a static JSP or HTML page into a portal component.

Note: The procedures listed in step 7 through step 11 are optional. You do not have to provide a configuration form for your portal components.

7 Right-click on the new portal component and select **New** > **Configure**.

A standard form page is added.

8 On the configure form, enter the relative URL to form you want to use to configure your portal component in the Alternative HREF field.

The URL should use the following format:

e_*moduleName_activtyName_formName*.do?*property*=value

For *moduleName* enter the module where your configuration form resides.

For *activityName* enter the activty name where your configuration form resides.

For *formName*, enter the name of your configuration form.

For *property*, enter any URL query you want to submit with the URL. This is an optional part of the URL and can be ignored if your configuration form does not require it. Typically, only DocExplorer forms require a property entry.

For example:

e_helpdesk_status_myPortalComp.do

9 Create a new form matching the Peregrine Studio address you entered in step 8.

This form will be the target configuration form of the Alternates HREF setting.

10 Define the following settings for your configuration form.

a Server OnLoad Script. This script must either be set to or call the portal.editComponent script.

Important: You can only have one server onload script per portal component that runs from either the contents or the configure form.

If you call the portal.editComponent function from a custom script you must adhere to the following script conventions:

■ Your custom script must include code similar to the following:

```
//Contents of your custom script

...

//Combine the messages from your function and portal.editComponent
Msg.add(yourMsg);
Msg = env.execute("portal.editComponent",Msg);
return Msg;
```

Do not return the Msg until after you have called portal.editComponent

■ Your custom script must preserve the value of the _Id variable that the portal.editComponent function passes to it.

b Save action. You must add a save action to your configuration form that has a target-form of portal.edit.save.

c No sidebar navigation. All of the out-of-the-box Get-Resources portal component configuration forms do not display in the navigation sidebar. If you want to follow this convention, then clear the Force display of sidebar menu option on the Others tab.

d Optional fields for configuration. All of the following fields are available from the portal.editComponent script. You can use them as document fields in your form components.

■ _column. Determines whether the component displays in a wide or narrow frame.

■ _title. The title you want to display for the portal component.

■ _originalTitle. The default name of the portal component that users can restore to. This field is typically used as a hidden field and should not be visible to users.

■ DtLastModify. The date the portal component was last modified to keep track opf changes or revisions. This field is typically used as a hidden field and should not be visible to users.

11 Add any additional form components you want to use to configure your portal component.

12 Save your project file.

13 Build your project and deploy your updated Get-Resources files to your application server's presentation folder.

Important: You must add an adapter name entry to the **Alias for** field in the **PortalDB** tab in order for Get-Resources to display portal components. This setting is available from the Administration page (admin.jsp).

Tailoring Get-Resources forms

The following sections describe how to tailor particular Get-Resources forms. In most cases, you can use personalization to add, remove, or change form content. Each section that requires manual tailoring has its own instructions.

Best Practices

The following general tips will enhance the ability to upgrade your project:

- Tailor the screens using personalization (the wrench icon) whenever possible.
- Avoid using studio to patch existing files. Get-Resources provides ways to extend the existing schemas and to locally change the product's behavior by deriving some scripts.

Changing the request summary screen

There are two main areas of this screen:

- The request detail information section (upper section).
- The list of selected items section (lower section).

Request detail information

List of selected items

You customize each area with a different method.

The request detail information section

Use personalization (the wrench icon) to change the display. If you do not find a field that you need with personalization, you must first add it to the Request schema. To display a subdocument (such as User information), you might have to extend the corresponding schema as well.

The list of selected items section

You cannot personalize this list, which means that you must use Peregrine Studio to make any changes.

Warning: This type of change will not upgrade automatically, and requires you to merge your changes during an upgrade. Therefore, you need to carefully weigh the need for a change in this area versus the ease of upgrade.

Changing all lists of selected items

Tailor the following component	To change
Project.Templates.newcart.newcatalog.doctable	Items listed in all request and purchase order summaries.

Changing individual lists of selected items

Tailor the following component	To change
Project.resources.request.build. requestsummary.newcart.newcatalog.doctable	Request checkout screen.
Project.resources.approve.approvedetail. requestsummary.newcart.newcatalog.doctable	**Show Approval List** activity.
Project.resources.request.requeststatus. requestsummary.newcart.newcatalog.doctable	Request summary in the **My submitted requests** and **My requests history** activity.

Schema Used
- RequestLine

By default, you can add any form component that uses a document field from the `RequestLine` schema. If you do not find the fields you want to display in this schema, you must create a schema extension to add the fields you want. In some cases, however, the field that you want to add is part of another schema, such as `Product`. In this case, you will want to create a script extension to merge the document fields between the two schemas `RequestLine` and `Product`. See *Example: adding a field from one schema to another schema* on page 185 for instructions on how to create a script extension to add a field to `RequestLine` from another schema.

You can add read-only or editable form components to this list from Peregrine Studio.

For an editable form component that you add to this list, you must create both a editable and a read-only version of the form component. Get-Resources determines which form component to display based the result of the form component's access field.

Step 1 Create an editable form component. See *Adding an editable form component* on page 158.

Step 2 Create a read-only version of the editable form component. See *Adding the read-only version of the editable form component* on page 158.

Step 3 Build and deploy your changes.

Adding an editable form component

To add an editable form component:

1 Create a new form component in Peregrine Studio.

2 Select the form component from the Project Navigator.

3 From the properties page, click the **Access** tab.

4 Enter the following values:

- **Access Field**. Enter `_bReadOnly`.
- **Access Value**. Leave empty.

Adding the read-only version of the editable form component

To add the read-only version of the editable form component:

1 Create a second identical form component in Peregrine Studio.

2 Select the form component from the Project Navigator.

3 From the properties page, click the **Access** tab.

4 Enter the following values:

- **Access Field**. Enter _bReadOnly.
- **Access Value**. Enter true.

Note: When creating a new request, the field that you added might be blank. If you want the default information pre-populated instead, see *Request line default values* on page 190.

Changing the request line detail screen

The Request line detail screen opens:

- When you look at the catalog item details.
- After clicking on **Configure** for a catalog item.
- When looking at the details of an item from the selected item list in the **Request summary** screen.
- When looking at the details of a subline item (from the Composition table in one of the screens previously listed).

You can personalize these screens and change their content using personalization. If you do not find the fields that you need on the personalization screen, you must first create a schema extension to add fields to the RequestLine schema.

Request line detail screens have more than one layout. The detail shown depends on two criteria:

- The line item **subtype**
  - bundle
  - off catalog
  - cable
  - work order
  - contract
  - training
  - ShopDirect
  - other
- The DocExplorer context

- Called from the catalog item list
- Called from the selected item list (from the **Request summary** page)
- Called from a subline item list (as part of a bundle)

One screen definition is saved for every combination of these two criteria. For example, a **Cable detail** screen can be configured differently when you select the detail from the catalog list than when you select it from the selected item list, or when you select it as part of a bundle (showing as a subitem). Likewise, a **Training detail** item can be personalized independently from a **Cable detail**.

Adding or removing subtypes from request line item details

You can add or remove subtypes from request line items by creating a custom ECMAScript function. You may use the existing function getLineItemSubType as a template. This function computes the subtype based on a line item's content.

The ECMAScript getLineItemSubType function for	Peregrine Studio address
ServiceCenter	Project.cartexperience.Scripts.requestinterfacebase. getLineItemSubType
AssetCenter	Project.resources.NewScripts.acrequestinterface. getLineItemSubType

To ensure an easier upgrade, you should create a custom request interface script to add or remove subtypes (see *Adding personalization* on page 171). Within your custom script, you can use any line item field listed in the RequestLine schema to determine the item's subtype.

If you want to add a subtype to the subtypes provided with Get-Resources, then your custom function needs to first check for new subtypes and then call the existing **getLineItemSubType** function to determine the out-of-box subtypes.

Type in your code here to determine the subtype based on the msgLineItem attribute values

Call the out-of-the box script if your code cannot determine the subtype

```
function getLineItemSubType(msgLineItem)
{
  var strSubType = "";
  // Try to determine your custom subtype here
  ...
  // If you did not find anything you were looking for in the
  // msgLineItem you can default to the parent behavior (shown here
  // for AssetCenter 4)
  if (strSubType == "")
    strSubType = ac4requestinterface.getLineItemSubType.apply
    (this, arguments);
  // Here, you can re-map the out-of-box subtypes that you
  // do not want any more, to another subtype. For example
  // if you do not care about the cable subtype:
  if (strSubType == "cable")
    strSubType == "catalogbase"
  return strSubType;
}
```

If you do not use the out-of-box subtypes in Get-Resources, you can modify your custom script by adding new subtypes to your **getLineItemSubType** function. You can use the following existing scripts as templates for subtypes.

Using this back-end	Use this script as a template
AssetCenter	**ac4requestinterface** or **ac3requestinterface**
ServiceCenter	**screquestinterface**

Changing the catalog select list

The Catalog select list screen opens:

■ When you look at the catalog item list.

■ When you look at the bundle list.

Catalog items in a bundle list

You cannot personalize this list, which means that you must use Peregrine Studio to make any changes.

Warning: This type of change will not upgrade automatically, and requires you to merge your changes during an upgrade. Therefore, you need to carefully weigh the need for a change in this area versus the ease of upgrade.

Changing all lists of selected items

Tailor the following component	To change
Project.Templates.newcatalog.doctable	■ All catalog screens that are used when building a new request or a new purchase order. ■ The list of selected items in the **Request summary** screen and **Purchase order summary** screen.

Changing individual lists of selected items

Tailor the following component	To change
Project.resources.request.build.itemlist. newcatalog.doctable	**Create a new request** activity.
Project.resources.approve.approvedetail.itemlist.newc atalog.doctable	**Show Approval List** activity.

Schema used

■ Product

By default, you can add any form component that uses a document field from the Product schema. If you do not find the fields you want to display in this schema, you must create a schema extension to add the fields you want.

Changing the purchase order summary screen

There are two main areas of this screen:

- The purchase order detail information (upper section).
- The list of selected items (lower section).

Purchase
order detail
information

List of
selected items

You customize each area with a different method.

The purchase order detail information section

Use personalization (the wrench icon) to change the display. If you do not find a field that you need on the personalization screen, you must first add it to the GRPurchaseOrder schema. To display a subdocument (such as User information), you might have to extend the corresponding schema as well.

The list of selected items section

You cannot personalize this list, which means that you must use Peregrine Studio to make any changes.

Warning: This type of change will not upgrade automatically, and requires you to merge your changes during an upgrade. Therefore, you need to carefully weigh the need for a change in this area versus the ease of upgrade.

Changing all lists of selected items

Tailor the following component	To change
Project.Templates.newcart.newcatalog.doctable	All request or purchase order summaries.

Changing individual lists of selected items

Tailor the following component	To change
Project.resources.buyer.createnewpo.requestsummary.newcart.newcatalog.doctable	The following activities: ■ **Create a new PO** ■ **My saved purchase orders in preparation** ■ **POs to review**
Project.resources.buyer.postatus.requestsummary.newcart.newcatalog.doctable	**My submitted purchase orders** activity.

Schema used

■ GRPOLine

By default, you can add any form component that uses a document field from the GRPOLine schema. If you do not find the fields you want to display in this schema, you must create a schema extension to add the fields you want.

You can add read-only or editable form components to this list from Peregrine Studio.

For an editable form component that you add to this list, you must create both an editable and a read-only version of the form component. Get-Resources determines which form component to display based the result of the form component's access field.

Step 1 Create an editable form component. See *Adding an editable form component* on page 166.

Step 2 Create a read-only version of the editable form component. See *Adding the read-only version of the editable form component* on page 166.

Step 3 Build and deploy your changes.

Adding an editable form component

To add an editable form component:

1 Create a new form component in Peregrine Studio.

2 Select the form component from the Project Navigator.

3 From the properties page, click the **Access** tab.

4 Enter the following values:

- **Access Field.** Enter _bReadOnly.
- **Access Value.** Leave empty.

Adding the read-only version of the editable form component

To add the read-only version of the editable form component:

1 Create a second indentical form component in Peregrine Studio.

2 Select the form component from the Project Navigator.

3 From the properties page, click the **Access** tab.

4 Enter the following values:

- **Access Field.** Enter _bReadOnly.
- **Access Value.** Enter true.

Note: When creating a new purchase order, the field that you added might be blank. If you want the default information pre-populated instead, see *Request validation* on page 208 and *Purchase order line default values* on page 214.

Changing the purchase order line detail screen

The **Purchase order line detail** screen opens:

- When you select an item's details or you click **Configure** from the **Select an item to add to the cart** screen (first screen of the **Create a new PO** activity).
- When looking at the details of an item from the selected item list in the **Purchase order summary** screen.
- When looking at the details of a subline item (from the Composition table in one of the screens previously listed).

You can personalize these screens and change their content using personalization. If you do not find the fields that you need on the personalization screen, you must first create a schema extension to add fields to the GRPOLine schema.

Purchase order line detail screens have more than one layout. The detail shown depends on two criteria:

- The line item **subtype**
  - bundle
  - off catalog
  - cable
  - work order
  - contract
  - training
  - ShopDirect
  - other.
- The DocExplorer context
  - Called from the **Select an item to add to the cart** screen
  - Called from the selected item list (on the **Purchase order summary** screen)
  - Called from a subline item list (as part of a bundle)

One screen definition is saved for every combination of these two criteria. For example, a **Cable detail** screen can be configured differently when you select the detail from the catalog list than when you select it from the selected item list, or when you select it as part of a bundle (showing as a subitem). Likewise, a **Training detail** item can be personalized independently from a **Cable detail.**

Adding or removing subtypes from purchase order line item details

You can add or remove subtypes from purchase order line items by creating a custom ECMAScript function. You may use the existing function **getLineItemSubType** as a template. This function computes the subtype based on a line item's content.

The ECMAScript getLineItemSubType function for	Peregrine Studio address
AssetCenter	Project.resources.NewScripts.acporderinterface.getLineItemSubType

To ensure an easier upgrade, you should create a custom request interface script to add or remove subtypes (see *Adding personalization* on page 171). Within your custom script, you can use any line item field listed in the **OrderLine** schema to determine the item's subtype.

If you want to add a subtype to the subtypes provided with Get-Resources, then your custom function needs to first check for new subtypes and then call the existing **getLineItemSubType** function to determine the out-of-box subtypes.

Type in your code here to determine the subtype based on the msgLineItem attribute values

Call the out-of-the box script if your code cannot determine the subtype

```
function getLineItemSubType(msgLineItem)
{
  var strSubType = "";
  // Try to determine your custom subtype here
  ...
  // If you did not find anything you were looking for in the
  // msgLineItem you can default to the parent behavior (shown here
  // for AssetCenter 4)
  if (strSubType == "")
    strSubType = acporderinterface.getLineItemSubType.apply
      (this, arguments);
  // Here, you can re-map the out-of-box subtypes that you
  // do not want any more, to another subtype. For example
  // if you do not care about the cable subtype:
  if (strSubType == "cable")
    strSubType == "catalogbase"
  return strSubType;
}
```

Changing the request line selection list

The request line selection list opens:

■ In the **Select an item to add to the cart** screen of purchase orders.

Request line select list—

You cannot personalize this list, which means that you must use Peregrine Studio to make any changes.

Warning: This type of change will not upgrade automatically, and requires you to merge your changes during an upgrade. Therefore, you need to carefully weigh the need for a change in this area versus the ease of upgrade.

Changing individual lists of selected items

Tailor the following component	To change
Project.resources.buyer.createnewpo.itemlist.newcatalog.doctable	**Create a new PO** activity

Schema used

- RequestLine

By default, you can add any form component that uses a document field from the RequestLine schema. If you do not find the fields you want to display in this schema, you must create a schema extension to add the fields you want.

Adding personalization

DocExplorers allow end users a means to create and customize searches of Get-Resources data. From the end-user perspective, personalization is a collection of standard forms that allow users to change part of the interface to suit their needs. The administrator determines which forms and features of personalization each user has by setting global personalization rights and by granting individual users capability words to do additional personalization.

From an application developer's perspective, a DocExplorer is a template activity that allows for the rapid development of Get-Resources changes without the need to rebuild a Peregrine Studio project for every change made. A DocExplorer enables you to add or remove fields, change the layout of a form, and change interface elements such as headers and buttons in real time using the browser interface.

Supporting personalization

Personalization of Get-Resources is provided in two ways:

- End-users can use personalization for all forms that have been built using Document Explorers (DocExplorers). Personalization allows authorized users to change the appearance and functionality of Get-Resources directly from the Web interface.

- Developers can use Peregrine Studio to add personalization capabilities to their own Get-Resources forms by creating new DocExplorers. This functionality can be enabled only by using Peregrine Studio.

To add Personalization capabilities to Get-Resources, you must have these components:

- An AssetCenter or ServiceCenter back-end database. Personalization requires you to store each user's login rights and personalization changes in a back-end database.

- Adapter aliases defined for the following tabs on the Get-Resources Administration settings page:

  - Portal
  - PortalDB

■ A user account with personalization rights enabled. A user's login profile determines the level of personalization rights Get-Resources grants to the user. A user's personalization rights determine not only what personalized components can be seen and changed, but also determines whether other users will see their personalization changes.

■ A configured DocExplorer activity to provide personalization in the Get-Resources Peregrine Studio project. You must configure each DocExplorer activity with an adapter name and a schema name. A DocExplorer can only use one schema at a time.

DocExplorer configuration required in Peregrine Studio

In order for users to use a DocExplorer from the Web interface, you must define at least two settings in Peregrine Studio:

■ The *schema* the DocExplorer uses. The schema determines what database tables and fields are available to query.

■ The *adapter* the DocExplorer uses to connect to the back-end database.

You can use any of the existing schemas provided with Get-Resources or create your own schema entries. For more information on schemas see *Document Schema Definitions* on page 107.

Adding a DocExplorer reference

A DocExplorer Reference is the preferred method for adding a DocExplorer to a Peregrine Studio project. A DocExplorer Reference is a special template that redirects users to a full DocExplorer activity with two parameters: the schema and adapter to be used. You can use a DocExplorer Reference to call any generic DocExplorer functionality.

To add a DocExplorer Reference:

1 Right-click on a Module component in your project. Select **New** > **DocExplorerReference**.

2 Enter a name for your new DocExplorer Reference activity. The default name is `DocExplorerReference`.

3 Expand the DocExplorerReference activity.

4 Click on the setup form.

5 On the form properties page, click the General tab and enter the following required information:

■ Title (en).

6 Select the redirect action.

7 On the properties page, click the Link Params tab.

8 Enter the parameters you want to use in the Param field. By default, this field has the following value:

```
_docExplorerContext=<DOCUMENT_NAME>&_DocExplorerBackend=<TARGET_NAME>
&_docExplorerSubType=<SUBTYPE_INSTANCE>
```

Replace <DOCUMENT_NAME> with the schema name you want the DocExplorer to use. This is a required parameter.

Replace <TARGET_NAME> with the adapter alias you want the DocExplorer to use. For example, enter ac for the AssetCenter adapter for Get-Resources. This is a required parameter.

Replace <SUBTYPE_INSTANCE> with the personalization form subtype you want to invoke or leave blank to use no subtype. This is an optional parameter.

Warning: Do not change the target form of the redirect action. This action must go to docExplorer.default.start.

9 Save your project.

10 Click the **Differential build of project** button to rebuild your project.

Personalizing a DocExplorer reference

After you have added a DocExplorer Reference, you can make changes to this activity directly from the Get-Resources Web interface.

To personalize DocExplorer pages:

1 Log in to Get-Resources.

2 Click the activity name for your Document Explorer from the navigation sidebar. By default, the Document Explorer will be called DocExplorer.

Important: The first time you access a Document Explorer, the interface will display a blank search form.

3 Click the wrench icon on the upper right of the interface.

4 Make your changes to the search form, and then click **Save**.

Your personalized search form is displayed.

5   Click **Search** to display the results list form.

6   Click the wrench icon from the upper right of the interface.

7   Make your changes to the list form, and then click **Save**.

8   Click on any of the results displayed in your personalized list form to go to the detail form.

9   Click the wrench icon from the upper right of the interface.

10 Make your changes to the detail form, and then click **Save**.

11 If you have user rights to create documents, click the activity name for your Document Explorer from the navigation sidebar to return to the search form.

12 Click **Create** to display the create form.

13 Click the wrench icon from the upper right of the interface. Make your changes to the create form, and then click **Save**.

Adding personalization form components – lookup fields

You can add special form components that use personalization features to your custom forms by adding lookup fields. These field components use some of the Personalization features found in a DocExplorer template activity.

Note: These form components are already part of the DocExplorer template activity and do not have to added. Lookup fields are intended to be added to custom forms that you have manually buily in Peregrine Studio.

You can add two types of lookup fields to your custom forms:

- Field Lookup. See *Field lookup* on page 174.
- Subdocument Lookup. See *Subdocument lookup* on page 176.

Field lookup

You can use the field lookup form component to select the value of one (and only one) particular field of a schema. The Lookup field queries the back-end database for all the values of a pre-defined field and displays values in a list. For example, when opening an asset request, you might want to create a lookup field for a Name field to list all the employee names in the back-end database.

To add a field lookup to a form:

1 Right click the form to which you want to add the lookup.

2 Go to **New** > **Field** > **Lookup**.

3 Enter the following settings for the Data attributes:

- Display Field—the label you want displayed for the lookup field in the Get-Resources form. If you do not enter a value for this parameter, the label defaults to the Document Field parameter described below.

- Document Field—the name of the field you want to use as the unique key for your query. In a field lookup, the value of this field must match the field name portion of the Document Path in step 4. This value will also be posted to the onload script when a particular lookup entry is selected.

4 Enter settings for the following DocExplorer Adapter attributes:

- Adapter—the name of the back-end database adapter you want to use to lookup the information.

- Document Path—the name of the schema and *field* that you want to lookup. The naming convention used with this parameter is *schema name.field name* with a period (.) between them. For example, the entry employee.name will lookup the name field from the employee schema.

5 Enter the following setting for the Link Parameters attribute:

- Target Form—enter docExplorer.fieldlookup.start as the form name. This value enables personalization if the end-user has sufficient personalization rights.

6 Click the **Differential build of project** button to rebuild your project.

7 Log in to Get-Resources, browse to the updated form, and click the magnifying glass lookup icon (🔍) to display a pop-up lookup form.

The lookup field displays a list of values that match the Document Path you entered in step 3 above.

8 If you want to change the field used for the lookup, click the **Personalize this page** link and select the new field you want to use.

Subdocument lookup

You can use a subdocument lookup form component to select all the field values that are part of a subdocument record (a subdocument typically has it's own schema). A subdocument lookup returns the value of each field defined in the external schema. Any other form components that use the information in the subdocument fields will be automatically updated. For example, you could use a subdocument lookup to update several fields such as address, state, zip, and country by selecting a single location.

Tip: Use subdocument lookups to quickly change multiple fields on a form.

To add a subdocument lookup to a form:

1 Right click the form to which you want to add the lookup.

2 Go to **New** > **Field** > **Lookup**.

3 Enter the following settings for the Data attribute:

- Display Field—the label you want displayed for the lookup field in the Get-Resources form. If you do not enter a value for this parameter, the label defaults to the Document Field parameter described below.

- Document Field—the name of the field you want to use as the unique key to query the subdocument. The value of this field is used to lookup all other document fields of the subdocument. This value is posted to the onload script when a particular lookup entry is selected.

4 Enter settings for the following for the DocExplorer Adapter attributes:

- Adapter—the name of the back-end database adapter you want to use to lookup the information.

- Document Path—the name of the schema and *subdocument* that you want to lookup. The naming convention used here is *schema name.subdocument name* with a period (.) between them. For example, the entry `employee.location` will lookup the location subdocument from the employee schema.

5 Enter the following setting for the Link Parameters attribute:

- Target Form. Enter `docExplorer.documentlookup.start` as the form name.

6 Click the **Differential build of project** button to rebuild your project.

7 Log in to your Web application, browse to the updated form and click the magnifying glass lookup icon ( ) to display a pop-up lookup form.

The lookup field will display a list of values that match the Document Path you entered in step 3 above.

8 If you want to change the subdocument used for the lookup, click the
   Personalize this page link and select the new subdocument you want to use.

Tailoring scripts

Although you do not have to use Peregrine Studio to edit or add scripts in your project, the text editor, cross reference checking mechanism, and project navigator make Peregrine Studio a full-featured development platform. The following sections describe how to change scripts from within Peregrine Studio.

Editing an existing script

You can edit the ECMAScript in your project directly from the Peregrine Studio interface.

Tip: You may lose changes that you make directly to existing scripts when you next upgrade. If you want to change an existing script consider using a schema extension to call your custom script in addition to the existing script.
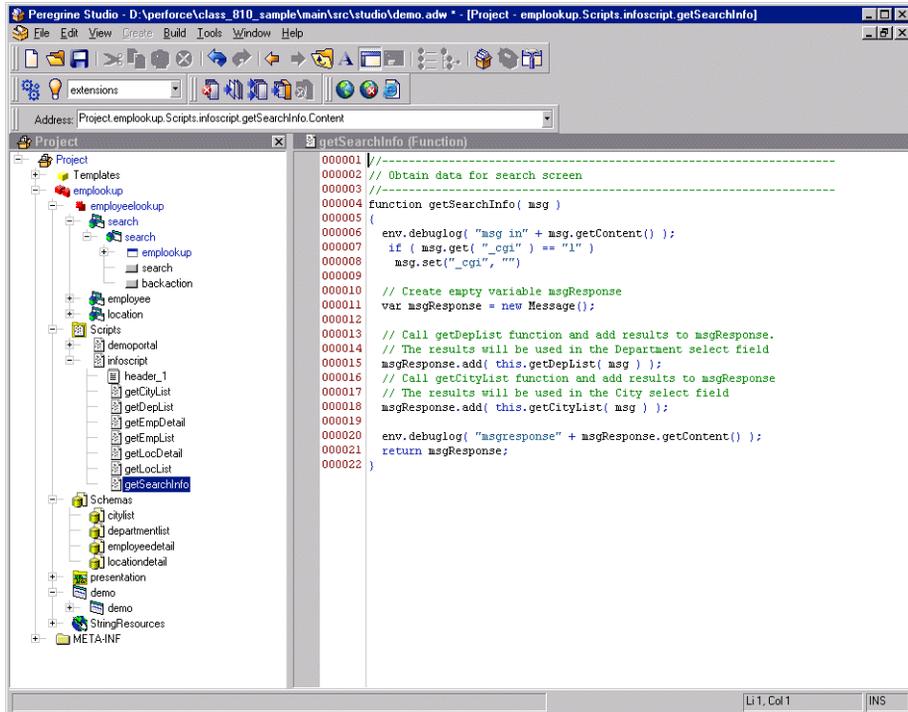
To edit an existing script:

1 Select the form in the Project Explorer.

2 Click the Script tab in the Properties window.
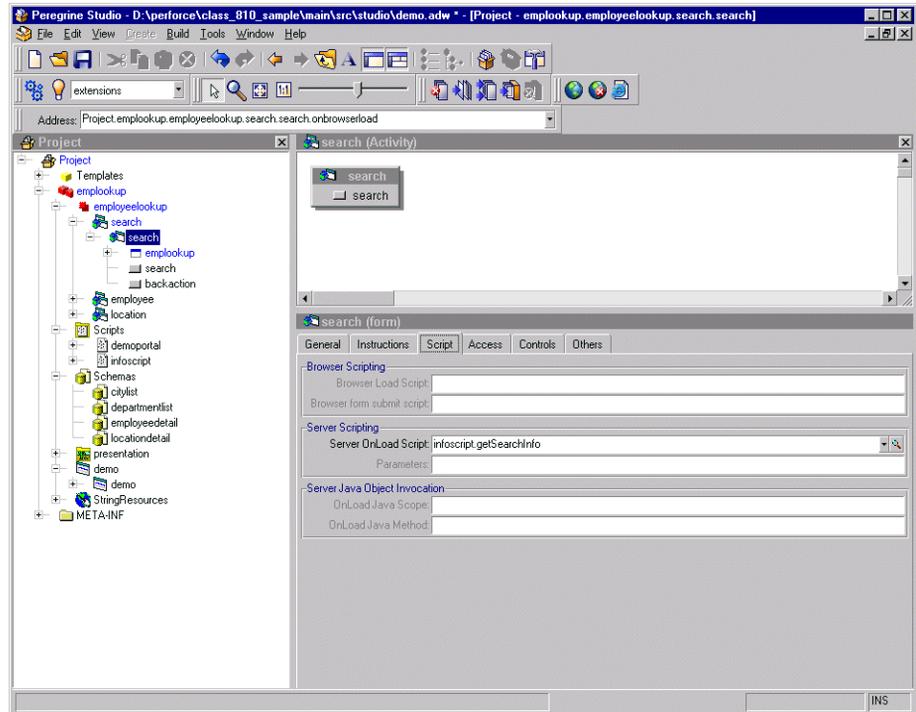
3 In the Server Onload Script field, click the magnifying glass button (🔍) to view the script in the Peregrine Studio text editor.

4 Make any changes to the script in the text editor.

5 Save your project.

6 Build your project file.

7 Restart your application server or set the **File Change Monitor** option from the Administration page.

> **Tip:** Turn off the File Change Monitor setting on your production system to increase performance.

The script update is loaded into Get-Resources.

Adding a custom script

You can add custom scripts to your Peregrine Studio project for use by forms, schemas, and form components.

To add a custom script:

1 Determine what kind of script you want to create.

You can create the following types of script:

- Form onload script. These are scripts run to gather data for non-DocExplorer forms. Peregrine Studio stores form on-load scripts underneath the first Group of Scripts node (Typically called **Scripts** or **ServerScripts**).

- Preexplorer. These are scripts run to manipulate the XML document that the gets rendered in the Get-Resources interface. Peregrine Studio stores preexplorer scripts underneath the **Preexplorer** Group of Scripts node.

- Preload. These are scripts run to gather data for DocExplorer forms. Peregrine Studio stores preload scripts underneath the **Preload** Group of Scripts node.

- Schema. These are scripts run before or after an adapter connects with the back-end database. Peregrine Studio stores schema scripts underneath the **Schema** Group of Scripts node.

2 Right-click the appropriate Group of Scripts node, point to **New**, and then click **Script**.

Peregrine Studio creates a new script node underneath the Group of Scripts.

3 Type in the name of your script and press ENTER.

4 Right-click the new Script node, point to **New**, and then click **Header**.

Peregrine Studio creates a new Header node underneath the Script node.

5 Using the text editor window, type in the header information for your new script.

6 Right-click the new Script node, point to **New**, and then click **Function**.

Peregrine Studio creates a new Function node underneath the Script node.

7 Using the text editor window, type in the function information for your new script.

8 Save your project.

9 Build your project file.

10 Restart your application server or set the **File Change Monitor** option from the Administration page.

The new script is loaded into Get-Resources.

Extending Get-Resources scripts

Using a script extension, you can override some of the out-of-box behavior without modifying the scripts that are shipped with Get-Resources. Using a script extension ensures that upgrades to later releases are easy by keeping your changes separate from existing Get-Resources functionality.

You can use script extensions to:

- Change the request behavior
- Determine how data is retrieved from the back-end database
- Determine how data is written to the database.
- Add or hide actions on a Get-Resources page
- Determine if data is displayed in read-only fields
- Create data validation rules
- Set default values

Changing request behavior

You can change the way the request and request line items work by creating a custom request interface script.

To change the request behavior:

1 Open the Get-Resources project file in Peregrine Studio.

2 Right-click a group of scripts node, and then click **New** > **Script**.

For example, you could select the following group of scripts:

Project.resources.NewScripts

Right-click on the NewScripts node and then click New > Script.
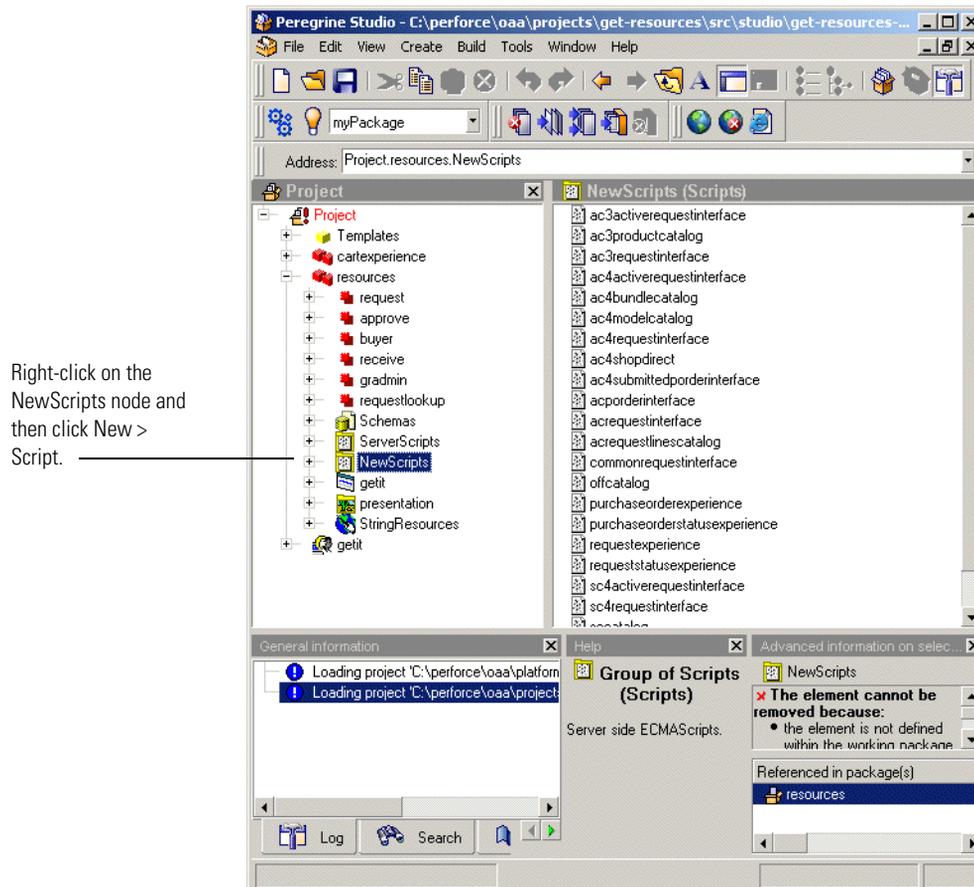
3 Give your script a unique name. For example, myrequestinterface.

4 Right-click your new script node, and then click **New** > **Header**.

You can accept the default Header name **Header**.

5 Use the following table to determine the script name that your custom script needs to import.

Back-end database	Script name
AssetCenter 3	ac3requestinterface
AssetCenter 4	ac4requestinterface
ServiceCenter 4 or 5	sc4requestinterface

6 Use the Peregrine Studio text editor to add a ECMAScript header that imports and makes a prototype of the script specific to your back-end database. For example:

Script name must match that used for your back-end database

```
import sc4requestinterface;
this.__proto__ = sc4requestinterface.valueOf();
```

7 Edit the existing **getRequestInterface** ECMAScript function to call your custom script name.

The existing script has the following Peregrine Studio address:

```
Project.resources.NewScripts.requestexperience.getRequestInterface.
```

You can delete checks for the back-end database versions from the **getRequestInterface** function since you already imported the necessary script for your back-end database version in a previous step. For example, your finished script might look like:

Return the name of your custom script

```
function getRequestInterface(msg)
{
    return "myrequestinterface";
}
```

8 Use the following table to determine the script name that your back-end uses to display the status of a request.

Back-end database	Script name
AssetCenter 3	ac3activerequestinterface

Back-end database	Script name
AssetCenter 4	ac4activerequestinterface
ServiceCenter 4 or 5	sc4activerequestinterface

9 Edit the display status script you selected above in Peregrine Studio to include a call to your custom script name. For example:

Script name must match your custom script name

```
import myrequestinterface;
this.__proto__ = myrequestinterface.valueOf();
```

10 You can now add the new functions to your custom script as described in the request sections. For example, *Changing the request line detail screen* on page 159.

You can copy the existing request interface functions into your custom script.

11 Save and build your Get-Resources project file.

Example: adding a field from one schema to another schema

If you want to display a field from a previous schema query in another schema, you can create a script extension to extract the value of a field from one document and insert it in another. For example, suppose you want to display the supplier part number you queried in a product summary in your purchase request summary. By default, the schema that builds request summary documents, RequestLine, does not contain a field to store or display the supplier part number. You can extend the RequestLine schema to add a field for supplier part number and then use a script extension to add the value you want from the Product schema.

To add a field from one schema to another schema:

1 Select a field from or add a field to the schema that you want to be the source of the information displayed.

For example, you can use a schema extension to add a field for supplier part number to the Product schema. This field queries the supplier part number as part of a product summary or product detail page. For more information on creating schema extensions, refer to the *Get-Resources Administration Guide*.

Schema extension logical mapping

This entry adds a field called SupplierPartNumber to the Product schema

```
<documents name="base">
  <document name="Product" label="Product">
    <attribute name="SupplierPartNumber" type="string"
      label="Supplier Part"/>
  </document>
</documents>
```

Schema extension physical mapping

This entry defines what field will be queried in your back-end database

```
<documents name="ac" version="4.1">
  <document name="Product" table="amCatRef">
    <attribute name="SupplierPartNumber" field="Ref"/>
  </document>
</documents>
```

2 Add a field to the schema that you want to be the target of the information to display.

For example, you can use a schema extension to add a field for supplier part number to the RequestLine schema. This field will store and save information on the supplier part number when it is updated by your script extension. For more information on creating schema extensions, refer to the *Get-Resources Administration Guide*.

Schema extension logical mapping

This entry adds a matching field to the RequestLine schema

```
<documents name="base">
  <document name="RequestLine" label="Req Line" ... >
    <attribute name="SupplierPartNumber" type="string"
      label="Supplier Part"/>
  </document>
</documents>
```

Schema extension physical mapping

This entry defines where the field will be saved in your back-end database

```
<documents name="ac" version="4.1">
  <document name="RequestLine" table="amReqLine" ... >
    <attribute name="SupplierPartNumber" field="CatalogRef.Ref"/>
  </document>
</documents>
```

Note: The <attribute> names do not have to match between the two schemas. This example uses matching field names to indicate that the fields store the same content.

3 Create a script extension to copy the value from the source field to the target field.

For example, you can create a script extension of the **catalogbase** script to read the value of the **Product** document's SupplierPartNumber field and add it to the **RequestLine** document's SupplierPartNumber field.

You can use the following information to create your script extension. For more information of creating script extensions, see *Adding personalization* on page 171.

Script setting	Value
Script to extend	catalogbase
Sample script extension	mycatalogbase

■ The **mycatalog** header must import the **sccatalog** script:

Import catalogbase ⎯⎯⎯⎯

```
import catalogbase;
this.__proto__ = catalogbase.valueOf();
```

■ Add a **getNewRequestLine** function to **mycatalogbase**.

```
function getNewRequestLine(msg)
{
  // Call the parent function to build the out-of-box request line
  // document
  var msgReqLine = catalogbase.getNewRequestLine.apply
  (this, arguments);
  // Read the value of a field from the Product schema and add it to
  // a field in the RequestLine schema
  msgReqLine.add( "SupplierPartNumber", msgItem.get
  ("SupplierPartNumber"));

  return msgReqLine;
}
```

Call the catalogbase script to create the request line document ⎯⎯⎯⎯

Adds the field value in ⎯⎯⎯⎯ MsgItem to the field value in msgRequestLine

**4** Use Peregrine Studio to add or configure a form component to display the value of the field.

For example, you can add a textedit form component to the
Project.Templates.newcart.newcatalog.doctable template and then map the form
component to the SupplierPartNumber field.

Changing purchase order behavior

You can change the way the purchase orders and purchase order line items
work by creating a custom request interface script.

1 Open the Get-Resources project file in Peregrine Studio.

2 Expand the **resources** group of modules node.

3 Expand the **NewScripts** node.

4 Right-click the **NewScripts** node, and then click **New** > **Script.**

Right-click on the
NewScripts node and
then click New >
Script.
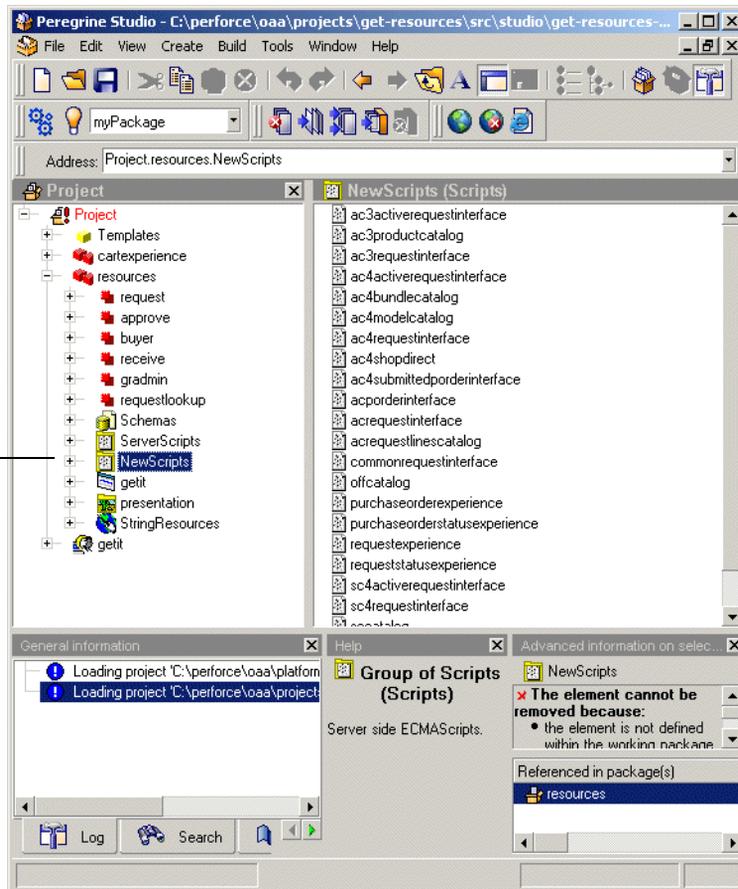
5 Give your script a unique name. For example, `myporderinterface`.

6 Right-click your new script node, and then click **New** > **Header**.

You can accept the default Header name `Header`.

7 Use the Peregrine Studio text editor to add a ECMAScript header that imports and makes a prototype of the **acporderinterface** script. For example:

Script name must be ———— `import acporderinterface;`
acporderinterface `this.__proto__ = acporderinterface.valueOf();`

8 Edit the existing **getRequestInterface** ECMAScript function to call your custom script name.

The existing script has the following Peregrine Studio address:

```
Project.resources.NewScripts.purchaseorderexperience.
getRequestInterface.
```

You can delete checks for the back-end database versions from the **getRequestInterface** function. For example, your finished script might look like:

```
function getRequestInterface(msg)
{
    return "myporderinterface";
}
```

Return the name of
your custom script————

9 Edit the existing **ac4submittedporderinterface** script in Peregrine Studio to make your custom script its prototype. For example:

Script name must ———— `import myporderinterface;`
match your custom `this.__proto__ = myporderinterface.valueOf();`
script name

10 You can now add the new functions to your custom script as described in the purchase order sections. For example, *Changing the purchase order line detail screen* on page 166.

You can copy the existing request interface functions into your custom script from one of the following scripts:

- **acporderinterface**

- **requestinterfacebase**

11 Save and build your Get-Resources project file.

Request line default values

You can set the request line default values in two places:

- As the user selects catalog entries
- In the `getRequestDefaultValues` function

Setting request line default values from catalog entries

Get-Resources generates a document using the `RequestLine` schema as a user selects items from the catalog. This document is updated with default values every time the user shows a catalog entry detail or clicks the Configure, Add, or Add Selected buttons. By setting the default values at these times, users can review the values before they decide whether to add the item to their request. This also allows Get-Resources to set default values that depend on the catalog entry without having to re-query the catalog information later.

The default values presented on the **Request line** screen are defined in the `getNewRequestLine` function of the `catalog` script. The `catalog` script can also display a filtered list of items based on the category, quick search, and advanced search criteria parameters entered.

The `catalogbase` script implements the basic catalog functionality. There are additional scripts that extend the `catalogbase` script to add more specific catalog functionality:

- `ac3productcatalog`
- `ac4bundlecatalog`
- `sccatalog`
- `offcatalog`.

To change the generated request line document:

Step 1 Identify the catalog script used to generate the request line detail you want to change. See *Identifying the catalog script that builds a request line detail* on page 191.

Step 2 Create a new script that extends the identified catalog script. See *Creating an extension of the existing catalog script* on page 191.

Step 3 Make Get-Resources call your new script. See *Calling your extended script* on page 193.

Identifying the catalog script that builds a request line detail

1 Enable the **show form info** setting from the **Administration** > **Settings** page.

2 Go to a request line detail screen. For example:

- Create a request
- Select a category
- Select one of the items on the catalog item list

3 Click the form information button.

The form information window opens.

4 Click the **Script Input** tab.

5 Search for the <CatalogId> element.

The value listed between the <CatalogId> elements is the name of the catalog script that generated the request line document.

Creating an extension of the existing catalog script

You can create a script extension to preserve the original script function provided with Get-Resources. This method improves the upgrade process for your installation.

Use the following information to create your script extension. For more information of creating script extensions, see *Adding personalization* on page 171.

Script setting	Value
Script to extend	sccatalog
Sample script extension	mycatalog

- The **mycatalog** header must import the **sccatalog** script:

Import sccatalog
```
import sccatalog;
this.__proto__ = sccatalog.valueOf();
```

■ Add a **getNewRequestLine** function to **mycatalog**.

```
function getNewRequestLine(msg)
{
  // Call the parent function to build the out-of-box request line
  // document
  var msgReqLine = sccatalog.getNewRequestLine.apply
  (this, arguments);
  // Set the default values you want. Here let's say that the default
  // requested quantity is 2 instead of 1
  msgReqLine.set("Quantity", "2", false);

  return msgReqLine;
}
```

Call the sccatalog
script to create the
request line document

Sets the Quantity field
to 2

The **RequestLine** schema defines the structure of the request line document. Refer to this schema to determine what fields are available for default values and format settings.

Calling your extended script

Use the following information to have Get-Resources call your script extension. For more information of creating script extensions, see *Adding personalization* on page 171.

Script setting	Value
Script to extend	sccatalog
Sample script extension	mycatalog
Functions to customize	getCatalogId
	getDefaultSearchCatalogId

■ Customize the **getCatalogId** and **getDefaultSearchCatalogId** methods:

```
function getCatalogId(msg)
{
  // Call the parent method
  var strCatalogId = sc4requestinterface.getCatalogId.apply
  (this, arguments);
  // Override the result only if the parent method returns sccatalog
  if (strCatalogId == "sccatalog")
    strCatalogId = "mycatalog";
  return strCatalogId;
}
```

Call your custom script if the catalog Id is sccatalog

```
function getDefaultSearchCatalogId(msg)
{
  // Call the parent method
  var strCatalogId = sc4requestinterface.getDefaultSearchCatalogId.
  apply(this, arguments);
  // Override the result only if the parent method returns sccatalog
  if (strCatalogId == "sccatalog")
    strCatalogId = "mycatalog";
  return strCatalogId;
}
```

Call your custom script if the search catalog Id is sccatalog

Overview of the cart experience code

The **cart experience** code allows you to do the following:

■ Create new requests

■ Create new purchase orders

■ Review a request details

■ Review purchase order details

- Display request line items
- Display purchase order line items.
- Select a request types
- Select item categories
- Select catalog items

The Get-Resources cart experience code is organized in four layers:

- The ActivityCartExperience template defines the screen flow.
- The cartexperience script controls the screen flow and checks that all of the fields, messages, and so on are passed to the screen. The actual data gathering and interactions with the back-end are handled by the requestinterface script.
- The requestinterfacebase script, and any scripts you create to extend it, are responsible for interacting with the back-end, implementing business rules, and describing the actions that are possible on requests and purchase orders. It is also responsible for listing what request categories, item categories, and catalogs are available.
- The catalogbase script, and any scripts you create to extend it, are responsible for retrieving the list of catalog entries and building a request or purchase order line upon request. The catalog scripts can be called by a request interface script or by the cartexperience script, however, the cartexperience script always gets the name of the catalog script from the request interface script.

The ActivityCartExperience template

You can create your own request activity using the CartExperience template.

To create a CartExperience activity:

1 Open your Get-Resources project in Peregrine Studio.
2 Expand the **resources** group of modules node.
3 Select a module to add the new activity to
4 Right-click the module and then click **ActivityCartExperience**.
   A new activity node appears underneath the selected module.
5 Rename the new activity.
6 Expand the new activity node and then expand the first form **setuprequest**.

7 Select the **start** action form component.

8 Click the **Link properties** tab from the properties page.

9 Click the **Param** attribute and locate the _cartExperience entry.

10 Replace the value <Set your cart experience script name here> with the name of your cart experience script. For example,

```
_cartExperience=mycartexperience
```

You can create a custom cart experience script that is similar to requestexperience, requeststatusexperience, or purchaseorderexperience scripts. Your custom cart experience script must contain an init function and a getRequestInterface function that returns the name of a request interface script (the name of the script that extends requestinterfacebase).

The cartexperience script

The cartexperience script generates and manages the data for the screens found in the ActivityCartExperience template. This script has at least one function for each screens in the ActivityCartExperience template.

The script is responsible to maintain the context information used in the cart experience activity. This context information is stored in an ECMAScript user object, one object per activity. The cartexperience.getCartSession method returns this context object for the current activity. The Message parameter passes _module and _activity elements that uniquely identify the current activity. These two parameters are always set in onload messages.

The main context object attributes are:

Attribute name	Attribute type	Description
strCartExperience	ECMAScript String	The name of the cart experience script declared in the setuprequest screen of ActivityCartExperience.
strRequestInterface	ECMAScript String	The name of the request interface script used to interact with the database in this activity.

Attribute name	Attribute type	Description
msgRequestCategory	Message	An XML document containing some information about the current request type. If a request category was selected it contains at least an Id attribute, and optionally a SubType attribute (that controls how the checkout screen personalization is saved).
msgRequestContent	Message	An XML document containing the request or purchase order document being edited. The document format depends on the schema that the request interface uses.
strApprovalId	ECMAScript String	The workflow task Id that the approver will either approve or deny. This string is set only when in the approval activity
strCategoryId	ECMAScript String	The id of the last item category that was selected in this activity.
strCatalogId	ECMAScript String	The name of the last catalog used to display the list of items. It is used mainly when the users click the **Add more items** button.
strCallingListForm	ECMAScript String	The form name (<module>.<activity>.<formname>) from which the current activity was called. It is used to go back to the caller screen, when you click the **Back to List** or **Discard Changes** button.
strCallingListParam	ECMAScript String	Parameters that need to be passed back to the caller screen.
msgCurrentLineItem	Message	An XML document containing the last request or purchase order line item for which details were presented. The document format depends on the schema that the request interface uses.

The request interface scripts

A request interface script is a script that extends the existing requestinterfacebase script.

There is only one request interface script used in a given activity.

In most functions, you can get the context object by calling cartexperience.getCartSession. You can store information for the current activity in this context object as needed. Just add your own parameters when needed.

The noticeable functions where the context object cannot be retrieved are the getRequestDefaultValues and validateRequest functions. They only take a request or purchase order message as a parameter.

The catalog scripts

A catalog script is a script that extends the existing catalogbase script.

A catalog script name is always retrieved through a request interface script, by calling the getCatalogId function. Before using this script, you need to call the request interface's getCatalogScript function, which loads the script in memory if needed.

There are three major functions in a catalog script:

Script	Function
getItemListStyle	Returns a constant that the cartexperience script uses to determine what style to display a selected catalog. The list style can be a list of items (the default view in catalogs) or a detail (as implemented in the offcatalog script). Other values are reserved for a future use.

Script	Function
getItemList	Returns a list of catalog items. This script must use the query parameters passed into the message when coming from the advanced search screen. It must also use the _searchText parameter passed in when performing a quick search. The quick search is the search box at the top of the item category and catalog list screens.
getNewRequestLine	Builds a request or purchase order line from a catalog item. This script must add all the needed sub-documents and, if necessary, add the item's composition. Every subline item must have its own Id, that can be set using the following formula:

env.getUniqueId() + "_" +
Math.round(10000*Math.random())

There are two special parameters that can be set in this function to change the way a line item or a subline item is saved:

- **DoNotSave**. If set to **true**, the function does not save the line item. This flag is set for example with AssetCenter 3.x on the subitems, because AssetCenter adds them automatically with the main line item.
- **DoNotSavePrices**. If set to **true**, the function does not save the prices stored in the document, but will let the back-end set its own default values when the line item is saved.

Creating custom schemas

You can create custom schemas to instruct the Archway Document Manager how to query, update, or insert information to your back-end databases. A custom schema gives you complete control over the logical and physical mappings used by your forms.

Tip: For most tailoring tasks, you can accomplish the same results using a schema extension. For more information on schema extensions, see the *Get-Resources Administration Guide*.

If you want to create custom schemas you will need to use Peregrine Studio to add the custom schema to your project and then to configure other project components to use the custom schema. Deploying a custom schema will also require building and copying project files to your Get-Resources server. The following procedures outline how to create a custom schema.

Step 1 Create or activate a package extension to save your changes in Peregrine Studio. See *Peregrine Studio project packages* on page 44.

Step 2 Add a new schema file to your Peregrine Studio project. See *Adding a schema to your Peregrine Studio project* on page 200.

Step 3 Add logical and physical mappings to your schema file. See *Adding logical and physical mappings to your schema* on page 200.

Step 4 Configure other project components to use your custom schema. See *Tailoring forms and components* on page 138.

Step 5 Rebuild your Get-Resources project. See *Building a project* on page 42.

Step 6 Deploy your new Get-Resources project files. See *Deploying tailoring changes* on page 49.

Adding a schema to your Peregrine Studio project

You can only add a custom schema to a *group of schemas* node. This node will also be a child element of a *group of modules* node, and typically has the name *Schemas*.

To add a schema to your Peregrine Studio project:

1 Right-click the group of schemas node to which you want to add a schema.

This node will be underneath the group of modules node for Get-Resources. If your project contains more than one group of modules, choose the one that has a group of schemas node.

2 Point to **New**, and then click **Raw Schema**.

A new node appears with the name Schema.

3 Rename your schema using the following conventions.

Schema Naming Conventions

Each custom schema you create should have a unique name to prevent data errors from naming conflicts. Your custom schema name should meet the following criteria:

- The schema name is unique from any other schema name in the Peregrine Studio project.
- The schema name is unique from any attribute name mapping within the schema.

Adding logical and physical mappings to your schema

After you have added a new schema to your Peregrine Studio project, you are ready to add logical and physical mappings. Studio displays the content of your custom schema in a text editor window. You can use the text editor window to review and edit the XML source code of your schema. You can also use any text editor to edit your schema.

Note: If you use an external text editor to edit your custom schema, Peregrine Studio will not pick up the changes until the next time you open the project file.

All schemas must have both a logical and a physical mapping section. The logical mapping section is where you define what names and labels Get-Resources uses for fields in the user interface. The physical mapping section is where you define what back-end database tables and fields are used by each logical mapping. The following sections describe how to create the logical and physical mapping sections.

Creating the logical mappings

Step 1 Add the XML namespace element and the two <schema> elements. See *Adding required schema elements* on page 201.

Step 2 Add two <documents> elements for the logical mappings. See *Adding logical mapping <documents> elements* on page 201.

Step 3 Add two <document> elements to define the schema name. See *Adding logical mapping <document> elements* on page 202.

Step 4 Add one <attribute> element for each logical mapping you want to create. See *Adding logical mapping <attribute> elements* on page 202.

Adding required schema elements

1 Add an <?xml> element to the top of the file:

```
<?xml version="1.0"?>
```

This element declares that the file uses the XML namespace.

2 Add two <schema> elements underneath the namespace declaration:

```
<schema>
</schema>
```

These elements notify the Archway Document Manager that this file is a schema. All schema definitions must be enclosed between these two elements.

Adding logical mapping <documents> elements

1 Add two <documents> elements between the <schema> element containers:

```
<documents>
</documents>
```

These elements are the container for the logical mappings.

2 Add the name attribute to the <documents> element:

```
<documents name="base">
```

The attribute value `name="base"` is required. This attribute value notifies the Archway Document Manager that this section is for logical mappings.

Adding logical mapping <document> elements

1 Add two <document> elements between the <documents> element containers:

```
<document>
</document>
```

These elements are the container for the schema document.

2 Add the `name` attribute to the <document> element:

```
<document name="schema_name">
```

For *schema_name*, enter the same name you selected when adding the schema to the Peregrine Studio project. This attribute value *must* match the file name of the schema (without the **.xml** extension) or an error will occur. The Archway Document Manager uses this attribute value to create an XML document of the same name.

Adding logical mapping <attribute> elements

1 Add one <attribute> element between the <document> elements for each logical mapping you want to create:

```
<attribute/>
```

Note: You can use the standard XML self-closing tag syntax <element /> with the <attribute> element. You can also close every <attribute> element with a </attribute> element if you want.

2 Add a `name` attribute to each <attribute> element:

```
<attribute name="sample"/>
```

The Archway Document Manager uses this attribute value to create an XML element in any document message built from this schema. For example, the Archway Document Manager would convert this attribute into the XML element <sample>.

3 Add a `type` attribute to each <attribute> element:

```
<attribute name="sample" type="string"/>
```

Get-Resources uses this attribute value to determine how to render the field in the user interface. For more information about the type attribute, see *<attribute>* on page 116.

4 Add any optional attributes to the <attribute> elements.

For more information about the attributes available for the <attribute> element, see *<attribute>* on page 116.

Creating the physical mappings

Step 1 Add two <documents> elements for each adapter you want to support. See *Adding physical mapping <documents> elements* on page 203.

Step 2 Add two <document> elements to define the back-end database table name. See *Adding physical mapping <document> elements* on page 204.

Step 3 Add one <attribute> element for each logical mapping you created. See *Adding physical mapping <attribute> elements* on page 205.

Adding physical mapping <documents> elements

1 Add another set of <document> elements between the <schema> element containers:

```xml
<?xml version="1.0"?>
<schema>
  <documents name="base">
    <document name="schema_name">
      <attribute name="Id" type="id"/>
      <attribute name="sample" type="string"/>
    </document>
  </documents>

  <documents>
  </documents>

</schema>
```

Add a second set of <documents> elements here

These elements are the container for the physical mappings.

2 Add the name attribute to the <documents> element:

```xml
<documents name="adapter_name">
```

For *adapter_name*, enter the abbreviation of the adapter you want to use to connect to your back-end database such as ac.

3 Add the version attribute to the <documents> element if you plan to add different physical mappings for each version of your back-end database:

```xml
<documents name="ac" version="4">
```

> **Important:** You can skip to the next section if you are not going to provide different physical mappings for multiple versions of your back-end database.

4 If you want to provide physical mappings for each version of your back-end database, repeat steps 1 through 3 for each version you want to support.

You must provide a different value for the version attribute for each set of <documents> elements.

Adding physical mapping <document> elements

1 Add another two <document> elements between the physical mapping <documents> element containers:

```xml
<?xml version="1.0"?>
<schema>
  <documents name="base">
    <document name="schema_name">
      <attribute name="Id" type="id"/>
      <attribute name="sample" type="string"/>
    </document>
  </documents>

  <documents name="ac">
    <document>
    <document/>
  </documents>

</schema>
```

Add a second set of <document> elements here

These elements are the container for the back-end database table to be queried.

2 Add the name attribute to the <document> element:

```xml
<document name="table_name">
```

For *table_name*, enter the SQL name of the table you want to map to. The Archway Document Manager uses this attribute value to query the back-end database table.

3 Add any optional attributes to the <document> element that you want to use to connect to the back-end database or to run process scripts.

For more information about the attributes available for the <document> element, see *<document>* on page 112.

Adding physical mapping <attribute> elements

1 Add one <attribute> element between the physical mapping <document>
elements for each logical mapping you created:

```
<attribute/>
```

Note: You can use the standard XML self-closing tag syntax <element /> with
the <attribute> element. You can also close every <attribute> element
with a </attribute> element if you want.

2 Add the identical name attribute to each <attribute> element as you defined in
the logical mappings:

```
<attribute name="sample"/>
```

Each logical mapping <attribute> element must have a matching physical
mapping <attribute> element. The Archway Document Manager uses this
value to determine which logical name maps to a particular back-end
database field.

3 Add a field attribute to each <attribute> element:

```
<attribute name="sample" field="field_name"/>
```

For *field_name*, enter the SQL name of the field you want to map to. The
Archway Document Manager uses this attribute value to query the back-end
database field.

4 Add any optional attributes to the <attribute> elements.

For more information about the attributes available for the <attribute>
element, see *<attribute>* on page 116.

Sample schema

The following is a sample schema that you can use for as a template for your own custom schemas.

XML namespace ——————

Logical mappings always use name="base"

Document name determines schema name. This schema is sample.xml

Physical mapping lists adapter name

Physical mapping uses same attribute elements

```xml
<?xml version="1.0"?>
<schema>

<!--================================================================
Logical Mappings: XML elements and data types defined
====================================================================-->
  <documents name="base">
    <document name="sample">
      <attribute name="Id" type="number"/>
      <attribute name="contact" type="string" label="Contact"/>
    </document>
  </documents>

<!--================================================================
Physical Mappings: Logical names mapped to SQL names
====================================================================-->
  <documents name="ac">
    <document name="sample" table="amRequest"/>
      <attribute name="Id" field="lReqId" />
      <attribute name="contact" field= "lEmplDeptId"/>
    <document/>
  </documents>

</schema>
```

Adding data validation

You can have Get-Services validate field values in one of two ways:

- Make an input field required. Users will not be able to submit a form until they have entered all required fields.
- Add a custom validation script or function. If you want to check the validity of the data users submit, you must create a validation script or function.

Making a field required

Personalization forms allows you to mark fields as required, forcing users to fill in a value for that field in order to proceed to the following page. You can use required fields to assure that users complete all the fields that ServiceCenter requires when creating an incident or problem ticket. For example, you can make every Call ticket (in the Service Management module) require a user to attach an asset to it.

To make a field required:

1 Login to Get-Resources with a user account that has getit.personalization.admin rights.

The user must have advanced personalization rights to save changes as default.

2 Navigate to the form you want to personalize, and then click the personalization wrench icon.

The Personalize Document Detail window opens.

3 From the Current Configuration window, double-click the field or subdocument that you want to require.

The field or subdocument properties window opens.

4 Select one of the following options:

- *Subdocument*. For a subdocument, select the **Required** option under the Explorer Options section.
- *Field*. For a field, toggle the **Required** option to **Yes**.

5 Click **Set as Default** to save your changes as the default view for all users.

All users who can see this form now see the required field or lookup.

Request validation

The **validateRequest** function validates user requests as part of the **acrequestinterface** script. Get-Resources calls this script before saving a request to the database from the **Request summary** screen. The only parameter is the document generated by the **Request** schema. The function must return the request message. Furthermore it must set an error condition and add an explanation to the user object if the request is not valid.

Peregrine recommends that you extend the **validateRequest** function with a custom script rather than updating the function directly. For more information of extending scripts, see *Adding personalization* on page 171.

Important: If you edit the **validateRequest** function directly, then you will need to maintain the request validation for any changes to the request schema and validation scripts that Peregrine makes in future versions of Get-Resources.

For example, the following sample adds a **validateRequest** function to the **myrequestinterface** custom script. This function checks to see if the user actually populated the **Purpose** field before saving. See *Adding personalization* on page 171 for more information about this sample custom script.

Check Purpose field for default value ⸺

Set user message to custom error message ⸺

Call the parent script for your back-end database

Set error condition if bValid is false ⸺

```
function validateRequest(msgRequest)
{
  var bValid = true;
  // Check that the purpose was actually set by the user
  if (msgRequest.get("Purpose", false) == "Enter Purpose")
  {
    // If purpose is default, then add an user message retrieved
    // from a string file
    user.addMessage(IDS.get("resources", "my_error_message"));
    bValid = false;
  }
  // Call the out-of-box (parent) script
  msgRequest = sc4requestinterface.validateRequest.apply
  (this, arguments);

  if (!bValid)
  {
    // If bValid is false, then set an error condition to prevent
    // Get-Resources from updating the database
    msgRequest.setCondition("error");
  }
  return msgRequest;
}
```

Purchase order validation

To validate the purchase order before it is saved, you can extend the **validateRequest** function defined in the **acporderinterface** script. See *Request validation* on page 208 for an example extension.

Assigning default values

Setting request default values

The default values presented on the **Request summary** screen are defined in the `getRequestDefaultValues` function of the `commonrequestinterface` script.

This script is called on a new request before presenting the **Request summary** screen, and also, as the request is saved for the first time in the database. This script's only parameter is the message generated from the **Request** schema. The function must return the full request document, with the default values set. It can modify the request document directly and send it back, or work on a copy of the request document and send the copy back.

It is this function's responsibility to make sure that a value is empty before setting a default value.

Warning: Failure to observe this rule could result in the function overwriting user entries.

Peregrine recommends that you extend the `getRequestDefaultValues` function with a custom script rather than updating the function directly. For more information of extending scripts, see *Adding personalization* on page 171.

While extending the `getRequestDefaultValues` function, implement the default values you want and call the out-of-box function to fill in any remaining default values. The advantages of this approach are:

- Less code to maintain on your end.
- Smoother upgrades for future releases.

Important: If you edit the out-of-the-box `getRequestDefaultValues` function directly, then you will need to maintain the default values for any new fields Peregrine adds to the request schema in future versions of Get-Resources.

For example, the following sample adds a **getRequestDefaultValues** function to the **myrequestinterface** custom script. This function changes the default values for the **Purpose** and **RequestedFor** fields. See *Adding personalization* on page 171 for more information about this sample custom script.

```
// Set the default values in the request according to the values
// already set in msgRequest
function getRequestDefaultValues(msgRequest)
{
  // Set the RequestedFor Date default two weeks from now
  if (msgRequest.get( "RequestedFor", false ) == "" )
  {
    var date = Calendar.getInstance();
    date.add(Calendar.DATE, 14);
    msgRequest.set("RequestedFor",
    DataFormatter.getArchwayDate(date.getTime().getTime()), false);
  }

  // Set the default purpose to upgrade
  if (msgRequest.get("Purpose", false) == "")
  {
    msgRequest.set( "Purpose", "Enter Purpose", false );
  }

  // Call the out-of-box (parent) script that will set the remaining
  // default values
  msgRequest = sc4requestinterface.getRequestDefaultValues.apply
  (this, arguments);

  return msgRequest;
}
```

Sets the RequestedFor date to 14 days

Sets the Purpose to Enter Purpose

Calls the parent script for your back-end database

Setting request line default values to values in a request

You can reuse any values entered in a prior request as default values in line item documents. For example, if you set the End User field in a request, you can re-use the value of this field for all line items that do not have another value explicitly defined.

Peregrine recommends that you extend the **getRequestDefaultValues** function with a custom script rather than updating the function directly. For more information of extending scripts, see *Adding personalization* on page 171.

While extending the `getRequestDefaultValues` function, implement the default values you want and call the out-of-box function to fill in any remaining default values. The advantages of this approach are:

- Less code to maintain on your end.
- Smoother upgrades for future releases.

For example, you can extend the **getRequestDefaultValues** function to
update the EndUser fields in the RequestLines collection based on the value
of the End User field in the request document.

Previous examples
truncated

Call to request default
values

Gather request values
for EndUserId and
EndUser

Set EndUserId to value
returned in message
request

```
function getRequestDefaultValues(msgRequest)
{
  ...
  // Call the out-of-box (parent) script that will set the remaining
  // default values
  msgRequest = sc4requestinterface.getRequestDefaultValues.apply
  (this, arguments);

  // Comment out the line below to enable the request line default
  // values if (false)
  {
    var strEndUser = msgRequest.get("EndUserId", false);
    var msgEndUser = msgRequest.getMessage("EndUser", false);
    // Get the message corresponding to the collection for Request
    // Lines
    var msgReqLines = msgRequest.getMessage("RequestLines", false);
    // Test if the collection exists
    if (msgReqLines)
    {
    // The collection exists, get the list of request lines
      var list = msgReqLines.getList("RequestLine", false);
      // Browse the request lines to set their default values
      for (var i = 0; i < list.getLength(); i ++)
      {
        // Get the request line for index i
        var msgReqLine = list.getMessage(i);
        // If there is no end user set, set it to the request's
        var strRLEndUser = msgReqLine.get("EndUserId", false);
        var strDefValRLEndUser = msgReqLine.get("_DefValEndUserId",
        false);
        if (  (strRLEndUser == "" || strRLEndUser ==
        strDefValRLEndUser) && strEndUser != strRLEndUser )
        {
          // set the end user id
          msgReqLine.set("EndUserId", strEndUser, false);
          msgReqLine.set("_DefValEndUserId", strEndUser, false);
          // set the EndUser subdocument, used to display the values.
          msgReqLine.remove("EndUser", false);
          if (msgEndUser)
            msgReqLine.add(msgEndUser);
        }
      }
    }
  }
  return msgRequest;
}
```

Assigning default values ◄ 213

Purchase order default values

To set the purchase order default value, you can extend the **getRequestDefaultValues** function defined in the **acporderinterface** script. See *Setting request default values* on page 210 for an example extension.

Purchase order line default values

You can set the purchase order line default values in two places:

- As the user selects approved request line items.
- As the purchase order default values are being set

Setting purchase order line default values as the user selects request line items

You can set default values on the purchase order default lines page that depend on the request lines. To do so, you must:

- Extend the **getNewRequestLine** function located in the **acrequestlinescatalog** script with a similar function in your own catalog script (for example, **myreqlinecatalog**). This function should return a GRPOLine document as defined in the **GRPOLine** schema.

- Extend the **getCatalogId** and **getDefaultSearchCatalogId** functions located in the **acporderinterface** script with similar functions in your own request interface script (for example, **myporderinterface**). You can have these functions return to your custom catalog script (for example, **myreqlinecatalog**).

For an example of how to extend these scripts and functions, refer to *Setting request line default values from catalog entries* on page 190.

Setting purchase order line default values with the purchase order values

You can set default values for purchase order line items that depend on the purchase order values. To do so, you must:

- Extend the **getRequestDefaultValues** function located in the **acporderinterface** script with a similar function in your own purchase order interface script. This function should check the documents returned by the OrderLines collection and set the default values based on the purchase order values.

For an example of how to extend these scripts and functions, refer to *Setting request line default values to values in a request* on page 211.

Translating tailored modules

Out of the box, all Get-It web applications are provided in English. You can order translated versions of Get-Resources by purchasing a language pack. Get-Resources 4.1 language packs are available in the following languages:

- French
- Italian
- German

Note: Refer to the Peregrine support web site to determine the current availability of Get-Resources language packs.

If you tailor your installation of Get-Resources, you will need to translate any strings that you added. The following sections describe how you can translate your tailored modules.

If you have a language pack version of Get-Resources, you will need to edit the existing string files for these applications and add any new strings that resulted from your tailoring efforts. For more information on the process, refer to *Editing existing translation strings files* on page 216.

If you do not have a language pack version of Get-Resources and you want to create a new translation, refer to the instructions in *Adding new translation strings files* on page 217.

To configure Get-Resources to use your new translation, refer to *Configure Get-Resources to use new string files* on page 218.

Editing existing translation strings files

You can make edits, additions, and deletions to string files outside of Peregrine Studio using any text editor or standard translation software.

To edit an existing translation string file:

1 Open the English string file for your Peregrine Studio project in a text editor or translation program.

   You can find all the translation string files in the application server's deployment directories:

   ■ *<application server install>*\webapps\oaa\WEB-INF\strings

   ■ *<application server install>*\webapps\oaa\WEB-INF\apps*<application group of modules name>*

   Note: The English string file will have the ISO-639 two letter abbreviation EN in the file name.

   All strings files have a STR file extension.

2 Search for any new text that you added to your tailored Peregrine Studio project.

   The string file uses the format illustrated below:

   String_label, "translated string"

   Where *String_label* is the Peregrine Studio name given to the string, and

   Where *translated string* is the actual value of the string to be translated.

   For example if you added a new button, you might look for:

   EMPLOOKUP_EMPLOYEELOOKUP_SEARCH_LABEL, "Search"

3 Copy the entire line containing the English string.

4 Open the string file for the target language in which you want to add a translation.

   Note: The string file will use the ISO-639 two letter abbreviation for the language in the file name.

5 Paste the copied English string into the target string file. You can paste the string at the end of the string file.

6 Change the "*translated string*" portion of the new string to the target language of your translation. For example, to change the string listed above to French, you might enter the following:

`EMPLOOKUP_EMPLOYEELOOKUP_SEARCH_LABEL, "Recherche"`

7 Save the new string file.

The new translation strings will be available as soon as you stop and restart the application server.

Adding new translation strings files

You can add new string files to provide additional language support to Get-Resources. The translation process can be accomplished using any text editor or standard translation software.

Important: Peregrine does not support any user translated versions of Get-Resources.

To edit an existing translation string file:

1 Open the English string file for your Peregrine Studio project in a text editor or translation program.

You can find all the translation string files in your application server's installation directory:

- *<application server install>*\webapps\oaa\WEB-INF\strings
- *<application server install>*\webapps\oaa\WEB-INF\apps*<application group of modules name>*

Note: The English string file will have the ISO-639 two letter abbreviation EN in the file name.

All strings files have a `STR` file extension.

2 Copy the entire the English string file.

3 Create a new string file for the target language in which you want to add a translation.

Note: The string file must use the ISO-639 two letter abbreviation for the language in the file name.

4 Paste the copied English string file into the new file.

5 Change the "*translated string*" portion of each string to the target language of your translation.

6 Save the new string file.

The new translation strings will be available as soon as you stop and restart the application server.

Configure Get-Resources to use new string files

1 Log in as an administrator (the administrator login page is located at admin.jsp).

2 Click **Settings**.

3 Click the **Common** tab.

4 Enter the two letter ISO-639 language code for the languages you want to support in the **Locales** field. The first code entered will be the default language used. The other languages you define will be available in a drop-down list.

5 In the **Content type encoding** field, enter the character encoding to be used for the display language. The following table lists some of the common character encoding formats.

Character Encoding	Character Set
ISO-8859-1	U.S. and Western European character sets. This is the default character set used by Studio.
Shift_JIS	Japanese character set
ISO-8859-2	Polish and Czech character set

6 Click **Save** at the bottom of the Settings form to save your changes.

7 On the Console form, click **Reset Server** to implement your changes.

Users will now be able to select the display language for their session used when they login to the Peregrine OAA Platform.

A Troubleshooting and FAQs

APPENDIX

This appendix contains troubleshooting information for Peregrine Studio and tailoring tasks.

This chapter covers the following topics:

- *Get-Resources Environment* on page 220
- *Peregrine Studio* on page 221
- *Scripting Errors* on page 224
- *Tailoring Errors* on page 226

Get-Resources Environment

This section describes warnings or errors that can be generated while running a Get-Resources in your system environment.

Out of memory error

Problem

Your application server has run out of memory resources.

Solution

Get-Resources run best on a system with a minimum of 512 MB of RAM. If you cannot add more physical memory to your machine, you can increase the virtual memory space used on your Windows system. Adding virtual memory will require more hard disk space and may degrade system performance as cached information is saved to and retrieved from the hard disk. Refer to your Windows help for information on setting or changing virtual memory.

Cannot start Java – JRE must be installed

Problem

Peregrine Studio produces an error message when you attempt to create a package or build a project.

```
Cannot start Java ('jvm.dll' not found). The JRE (Java Runtime
Environment) must be installed ...
```

Solution

Install a dedicated copy of the Java 2 SDK for Peregrine Studio to use. You can install the Java 2 SDK from the Get-Resources Tailoring Kit installation CD.

Peregrine Studio

This section describes common problems with write protections, conflicts, and build errors generated with Peregrine Studio.

Cannot edit — components are displayed with grey background

Problem

Peregrine Studio displays some or all of your project components with a grey background, and you cannot make or save changes to the project components.
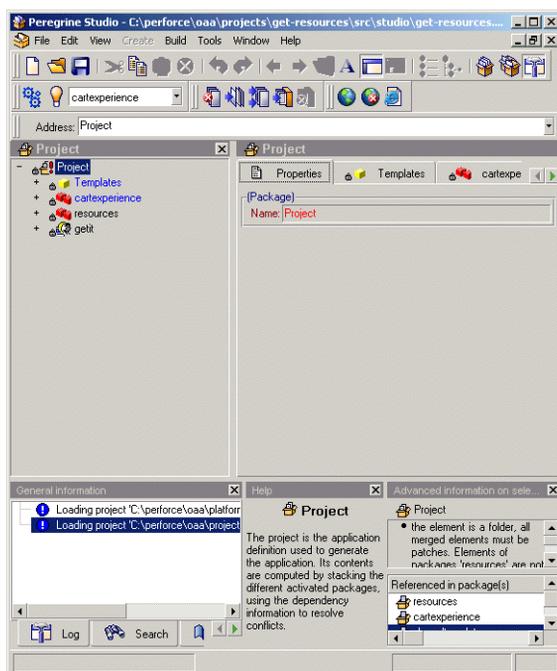
Solution

Peregrine Studio uses the grey background to indicate that an item is write protected. The most common reasons that Peregrine Studio components are write protected are:

- A write-protected package is selected in the package selector.
- The project (.adw) file is set to read-only.

Packages delivered by Peregrine Systems are write-protected. You must save all of your changes and additions to a user-created package extensions. If the package selection box displays one of the Peregrine Studio default packages, then your project will be write protected until you create and activate a new package extension in which to save your changes.

Red exclamation point (conflict icon) displayed next to nodes

Problem

Peregrine Studio displays a conflict icon next to one or more of your project components, and you cannot build the project. The conflict could be the result of multiple packages attempting to change or modify the same component, or the conflict could be the result of improperly defined package dependencies.

Solution

To resolve the conflict you should first view more information about the nodes displaying the conflict icon.

To view information about a conflict:

1 Select a node with an exclamation point ❗ icon displayed next to the name from the Project Explorer view.

2 Click **View** > **Advanced Information**. ProductCoreAbbreviated displays a new information window at the bottom of the interface. This window displays information on the conflict.

The information on selection will tell you whether you have a resource or a dependency conflict.

Conflict icon

Information about the conflict
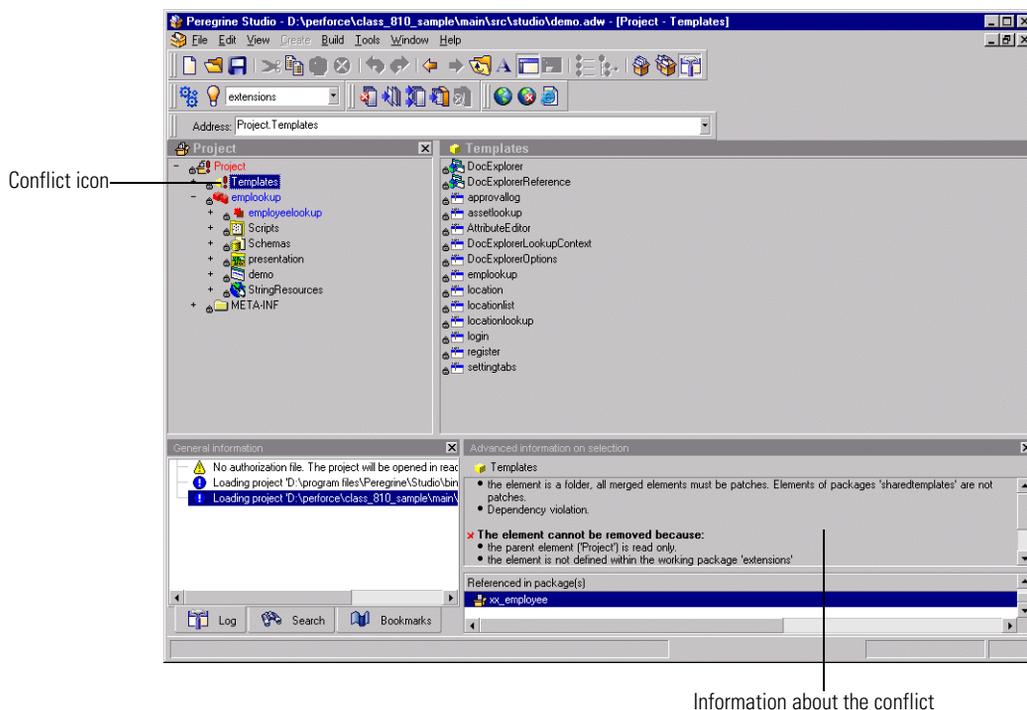
Resource conflicts

Resource conflicts occur when two or more project components describe the same thing. To resolve a resource conflict, delete or reconfigure one of the project components that is creating the conflict. If the conflicting components are part of separate package extensions, you can choose to deactivate one of the package extensions to resolve the conflict.

Dependency conflicts

Dependency conflicts occur when a package extension attempts to modify a package that is not listed as a dependent package. To resolve the conflict you can choose one of two solutions:

■ Add the package you want to modify as a package dependency of the conflicting package extension.

■ Move the changes in the conflicting package extension to another package extension that already has the proper package dependencies.

Scripting Errors

Information about scripting errors is displayed as text at the top of the main frame and in the archway.log file.

Unable to find script file

Problem

The following error message is displayed when you select a form:

Unable to find script file for <name>

This message will also appear in the archway.log file.

Solution

This error message is usually the result of a script file trying to call an undefined adapter. This is a common problem if you import a Web application into a project that contains a new adapter. Review your script file and determine what adapters it calls. If the <name> value is the name of a new adapter defined in the script file, then define the new adapter in the Admin Settings module, stop and restart your application server, and then restart the Archway server (using the Admin Control Panel) to correct the problem.

This error message can also appear if a form is calling an invalid script file name. Verify in ProductCoreAbbreviated that the form is calling a valid script file name. If you copied a script from another form or Web application you may have renamed the script incorrectly.

If you have verified that the script file exists and uses the proper adapter, then stop and restart your application server. This will refresh the adapter settings.

Script produces an ECMAScript error

Problem

An ECMAScript Error is displayed with the script name, source code, and line number of the error when a form is displayed.

Solution

Open Peregrine Studio, review the error-producing script for typos, and verify that it uses the correct function and schema names. For example, you might have a function where *msg* is incorrectly listed as *nsg*. Correct any errors and rebuild the project.

Note: ECMAScript is case sensitive and will return an error message if the case does not match the object called.

Tip: If you have enabled the HTTP listener in Peregrine Studio, you can click on the underlined script name listed at the top of the error message to go directly to the script and line number of the error. Peregrine Studio must be open for the hyperlink to work.

ECMAScript error: undefined value or property

Problem

The following error is displayed when you select a form:

ECMAScript Error: Error Message: Runtime error Function called on undefined value or property

This error will also be displayed in the archway.log file.

Solution

Verify that the form calls the proper script name in the server onload script attribute. Also check that the script name contains no typos and that it is listed with the proper case. If the script name listed in the form is correct, there is a possibility that there is a script name conflict. Each script in your project needs a unique name. Try renaming your script to a new name, updating the server onload script attribute, and rebuilding your project. If renaming the script fixes the problem then you had a script name conflict.

Tailoring Errors

The following sections describe some of the common errors associated with tailoring Get-Resources . Refer to the sections below for solutions to common tailoring problems.

Script output not appearing in form component

Problem

Data is not displayed in your Get-Resources form component. This problem could be the result of a faulty script that is not generating an XML document or the result of form components that are not properly mapped to the fields of the generated XML document.

Solution

Verify whether your script is generating an XML document by enabling the **Show form information** option and then looking at the contents of the Script Output tab. If the script is working properly, you should see your Get-Resources data encoded as in the XML document displayed on the Script Output page. If you do not see an XML document, then your script has an error.

If you can see data displayed in the Script Output tab, then the problem is how you have mapped the form components to the XML fields. View the form component properties from Peregrine Studio, and verify that the Document Field attribute of the form component maps to an XML tag displayed in the Script Output tab.

Too few parameters error

Problem

The following error message is displayed when you select a form:

ERROR:....: ***SQL Exception caught***

The script output displays the following error:

-3010: [...][...] Too few parameters. Expected 1.

These messages will also appear in the archway.log file.

Solution

There is an incorrect field mapping or typo in the schema used in this form. Review the schema(s) used by this form and verify that there are no typos. Also verify that all the attributes defined in the schema map to valid fields in the back-end database. The value in the field attribute must match the field name of the back-end database. This is particularly important for the ID attribute, which must map to a unique numerical value that identifies each record.

Get-Resources always goes to redirection form

Problem

You have defined a redirection to another form in Get-Resources and the source form always takes users to the redirection form regardless of the search conditions and results.

Solution

Validate that the Condition attribute of the redirection is not blank. The Condition value should match the value defined by the setCondition function of your form's ECMAScript. If the Condition attribute is left blank, the default action is to redirect to the target form regardless of the returned results.

Syntax error in FROM clause

Problem

The following error message is displayed when you select a form:

ERROR:...: ***SQL Exception caught***

The script output displays the following error:

-3506 [...][...] Syntax error in FROM clause.

This error will also be displayed in the archway.log file.

Solution

The schema name you defined for the form is wrong. The schema name could be listed incorrectly in two places:

- The form's onload script may refer to the wrong schema name.
- The <document name=*value*> does not match the schema file name.

Index