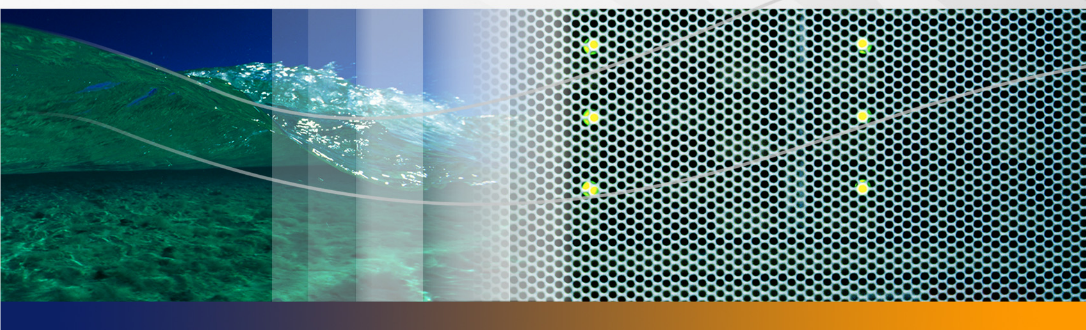
Product Documentation

Peregrine Systems, Inc.

Get-Resources™ 4.2



Tailoring Kit Guide

Part No. DGR-42-EN27

© Copyright 2005 Peregrine Systems, Inc.

PLEASE READ THE FOLLOWING MESSAGE CAREFULLY BEFORE INSTALLING AND USING THIS PRODUCT. THIS PRODUCT IS COPYRIGHTED PROPRIETARY MATERIAL OF PEREGRINE SYSTEMS, INC. (“PEREGRINE”). YOU ACKNOWLEDGE AND AGREE THAT YOUR USE OF THIS PRODUCT IS SUBJECT TO THE SOFTWARE LICENSE AGREEMENT BETWEEN YOU AND PEREGRINE. BY INSTALLING OR USING THIS PRODUCT, YOU INDICATE ACCEPTANCE OF AND AGREE TO BE BOUND BY THE TERMS AND CONDITIONS OF THE SOFTWARE LICENSE AGREEMENT BETWEEN YOU AND PEREGRINE. ANY INSTALLATION, USE, REPRODUCTION OR MODIFICATION OF THIS PRODUCT IN VIOLATION OF THE TERMS OF THE SOFTWARE LICENSE AGREEMENT BETWEEN YOU AND PEREGRINE IS EXPRESSLY PROHIBITED.

Information contained in this document is proprietary to Peregrine Systems, Incorporated, and may be used or disclosed only with written permission from Peregrine Systems, Inc. This book, or any part thereof, may not be reproduced without the prior written permission of Peregrine Systems, Inc. This document refers to numerous products by their trade names. In most, if not all, cases these designations are claimed as Trademarks or Registered Trademarks by their respective companies.

Peregrine Systems, AssetCenter, AssetCenter Web, BI Portal, Dashboard, Get-It, Peregrine Mobile, and ServiceCenter are registered trademarks of Peregrine Systems, Inc. or its subsidiaries.

Microsoft, Windows, Windows 2000, SQL Server, and names of other Microsoft products referenced herein are trademarks or registered trademarks of Microsoft Corporation. This product includes software developed by the Apache Software Foundation (http://www.apache.org/). This product also contains software developed by: Sun Microsystems, Inc., Netscape Communications Corporation, Javier Iglesias, and InstallShield Software Corporation. If additional license acknowledgements apply, see the appendix of the Installation Guide.

The information in this document is subject to change without notice and does not represent a commitment on the part of Peregrine Systems, Inc. Contact Peregrine Systems, Inc., Customer Support to verify the date of the latest version of this document. The names of companies and individuals used in the sample database and in examples in the manuals are fictitious and are intended to illustrate the use of the software. Any resemblance to actual companies or individuals, whether past or present, is purely coincidental. If you need technical support for this product, or would like to request documentation for a product for which you are licensed, contact Peregrine Systems, Inc. Customer Support by email at support@peregrine.com. If you have comments or suggestions about this documentation, contact Peregrine Systems, Inc. Technical Publications by email at doc_comments@peregrine.com. This edition of the document applies to version 4.2 of the licensed program.

Peregrine Systems, Inc.
3611 Valley Centre Drive San Diego, CA 92130
858.481.5000
Fax 858.481.1751
www.peregrine.com

Contents

PEREGRINE

Introducing the Get-Resources Tailoring Kit. 11

About this guide 12

Conventions used in this guide 13

Need more information? 14

Customer Support 14

Documentation Web site 14

Education Services Web Site 15

Chapter 1 Installing on Windows 17

Installing the Get-Resources Tailoring Kit 17

Upgrading the Get-Resources Tailoring Kit 22

Opening the Get-Resources project. 23

Setting up a tailoring environment 24

Setting up a development environment 24

Setting up a testing environment. 25

Chapter 2 Using Studio 27

The Peregrine Studio interface 27

Project Explorer 29

Drag and drop 31

Best practices 32

Do not change form definitions outside Peregrine Studio 32

Avoid enabling advanced options 33

Avoid using the clean the target folders build option 33

Clear application server cache 33

Use templates to apply global changes 33

Enable the HTTP listener and form information options 34

Set the color for your extension changes 36

View referenced components with the lookup button 37

Chapter 3 Peregrine Studio Projects and Packages 39

Peregrine Studio projects 39

Project components 40

Project component descriptions 41

Project files 43

Building a project 45

Build options 45

Setting project build settings 46

Peregrine Studio project packages 48

Saving changes with package extensions 48

Activating and deactivating packages 49

Package dependencies 50

Setting package dependencies 51

Warnings for conflicts 52

Deploying tailoring changes 53

Deploying to Windows platforms 54

Deploying to UNIX platforms 54

Chapter 4 Studio Components 55

Adding components 56

Types of form components 67

Component template containers 67

Fieldsection containers 68

Text edit fields 70

Selectbox fields 71

Hidden data fields 73

Redirections 74

Simple table 75

Document table 77

Table links 78

Text columns 79

Form columns 80

Actions 81

Chapter 5 Scripting 83

Overview of scripts 83

Script types 84

- Where to store scripts . . . 84
- How to use scripts . . . 85
- Editing an existing script . . . 87
- Adding a custom script. . . . 89
- Date values in scripts. . . . 91

Testing scripts . . . 92

- Rhino JavaScript debugger . . . 92
- URL queries . . . 94

Common message operations . . . 97

Using ECMAScript in an object-oriented manner . . . 100

- ECMAScript implementation in Get-Resources . . . 100
- Name resolution in ECMAScript. . . . 100
- Using the object prototype for object-oriented programming . . . 101
- How to use object orientation for tailoring . . . 105

Sample scripts . . . 105

- General script samples . . . 105
- Selecting a field from a schema . . . 106
- Calling other scripts and combining the results . . . 107
- Form script sample. . . . 109
- Creating an XML document from a schema . . . 109
- Working with dates in scripts . . . 112

References . . . 113

- Sources for client-side JavaScript . . . 113
- JavaDocs for the main Archway package . . . 114

Chapter 6 Tailoring Tasks 115
Tailoring workflow 116
List of tailoring tasks 117
Forms and form components 117
DocExplorers. 118
Scripting 118
Schemas 118
Data validation. 118
Default values 119
Translation. 119
Tailoring forms and components 119
Changing a form's title 120
Changing a form's instructions 121
Changing a form's onload script 122
Changing a form component's label 123
Hiding a form component 124
Changing a form component to read-only 125
Changing the schema that a form component uses 126
Changing the document field that a form component uses . . . 127
Displaying a form within a frameset. 130
Adding Get-Resources to an existing frameset. 132
Displaying a script variable in a form component 133
Creating a portal component 134

Tailoring Get-Resources forms 138
Best Practices 138
Changing the request summary screen 139
Changing the request line detail screen. 142
Changing the catalog select list. 144
Changing the purchase order summary screen 146
Changing the purchase order line detail screen 149
Changing the request line selection list 151
Adding personalization 152
Supporting personalization 152
DocExplorer configuration required in Peregrine Studio 153
Adding a DocExplorer reference 153
Personalizing a DocExplorer reference 155
Adding personalization form components – lookup fields 156
Tailoring scripts 159
Editing an existing script 159
Adding a custom script. 162
Extending Get-Resources scripts 163
Changing request behavior 163
Example: adding a field from one schema to another schema . . 165
Changing purchase order behavior 168
Request line default values 170
Setting request line default values from catalog entries 170
Overview of the cart experience code. 173

ActivityCartExperience template 174
The cartexperience script. 175
The request interface scripts 176
The catalog scripts 177
Creating custom schemas 178
Adding a schema to your Peregrine Studio project 179
Adding logical and physical mappings to your schema 180
Sample schema 187
Adding data validation. 187
Making a field required. 187
Request validation 188
Purchase order validation 190
Assigning default values 190
Setting request default values 190
Setting request line default values to values in a request. 192
Purchase order default values 194
Purchase order line default values 194
Translating tailored modules 195
Editing existing translation strings files 195
Adding new translation strings files. 198
Configure Get-Resources to use new string files 199

Chapter 7 Troubleshooting and FAQs 201
Get-Resources environment 202

Out of memory error 202
Cannot start Java – install JRE. 202
Peregrine Studio. 203
Cannot edit — components have grey background 203
Red exclamation point displays next to nodes. 204
Scripting errors 206
Unable to find script file 206
Script produces an ECMAScript error 206
ECMAScript error: undefined value or property 207
Tailoring errors 207
Script output not appearing in form component 208
Too few parameters error 208
Get-Resources always goes to redirection form 209
Syntax error in FROM clause 209
Appendix A Copyright Notices 211
Notices 211
Index 223

PEREGRINE

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 definitions 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 definitions 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 definitions that define the functionality of Get-Resources. The Get-Resources XML form definitions 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 developers who tailor Get-Resources from the source code provided with the tailoring kit.

Use this guide in conjunction with the following:

- 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.

This guide uses the following documentation conventions.

Text Formatting	Meaning
Bold	Information that you must type exactly as shown appears in bold. The names of buttons, menus, and menu options also appear in **bold**.
Italics	Variables and values that you must provide are in *italics*. New terms are in *italics*.

`Monospace`	Code or script examples, output, and system messages are in a `monospace` font.
	`var msgTicket = new Message( "Problem" );`
	…
	`msgTicket.set( "_event", "epmc" );`
	
	An ellipsis (…) indicates omitted portions of a script. Samples of code are not entire files, but they are representative of the information discussed in a particular section.
	Filenames, such as `login.asp`, appear in a `monospace` font.

Need more information?

For further information and assistance with this release, you can download documentation or schedule training.

Customer Support

For further information and assistance, contact Peregrine Systems' Customer Support at the Peregrine CenterPoint Web site.

To contact customer support:

1 In a browser, navigate to *http://support.peregrine.com*

2 Log in with your user name and password.

3 Follow the directions on the site to find your answer. The first place to search is the KnowledgeBase, which contains informational articles about all categories of Peregrine products.

4 If the KnowledgeBase does not contain an article that addresses your concerns, you can search for information by product; search discussion forums; and search for product downloads.

Documentation Web site

For a complete listing of current Get-Resources documentation, see the Documentation pages on the Peregrine Customer Support Web.

To view the document listing:

1 In a browser, navigate to *http://support.peregrine.com*.

2 Log in with your login user name and password.

3 Click either Documentation or Release Notes at the top of the page.

4 Click the Get-Resources link.

5 Click a product version link to display a list of documents that are available for that version of Get-Resources.

6 Documents may be available in multiple languages. Click the Download button to download the PDF file in the language you prefer.

You can view PDF files using Acrobat Reader, which is available on the Customer Support Web site and through Adobe at *http://www.adobe.com*.

Important: Release Notes for this product are continually updated after each release of the product. Ensure that you have the most current version of the Release Notes.

Education Services Web Site

Peregrine Systems offers classroom training anywhere in the world, as well as “at-your-desk” training using the Internet. For a complete listing of Peregrine’s training courses, refer to the following web site:

http://www.peregrine.com/education

You can also call Peregrine Education Services at +1 858.794.5009.

1 CHAPTER Installing on Windows

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

Before you begin the installation, you should have already installed Get-Resources.

This chapter covers the following topics:

- Installing the Get-Resources Tailoring Kit on page 17
- Upgrading the Get-Resources Tailoring Kit on page 22
- Opening the Get-Resources project on page 23
- Setting up a tailoring environment on page 24

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.

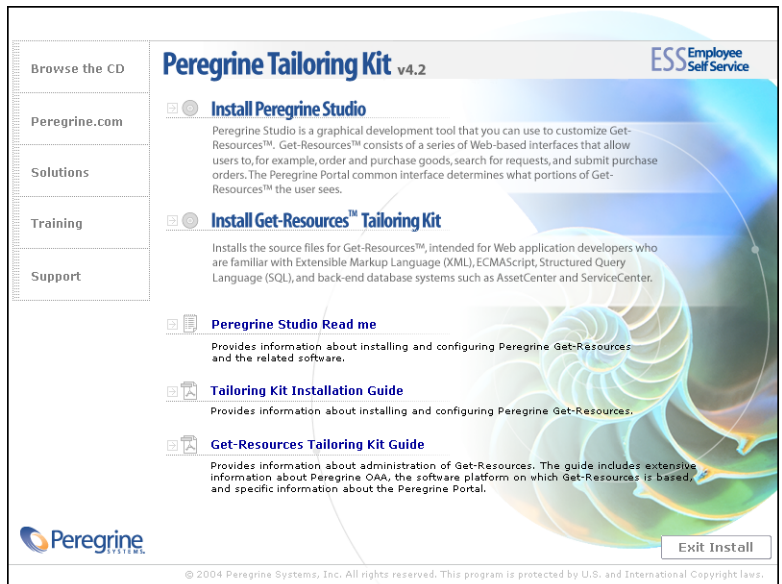
To install the Get-Resources Tailoring Kit:

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

   If you are installing on a system that has autorun enabled, the CD browser starts automatically. If autorun is disabled, you can manually start the CD browser.

   - Use Windows Explorer to navigate to the CD-ROM directory. Double-click **autorun.exe**.
   - Start the Get-Resources installation from the Windows command prompt. Type **D:\>autorun.exe** where `D` identifies the CD-ROM drive. Substitute your CD-ROM drive identifier.

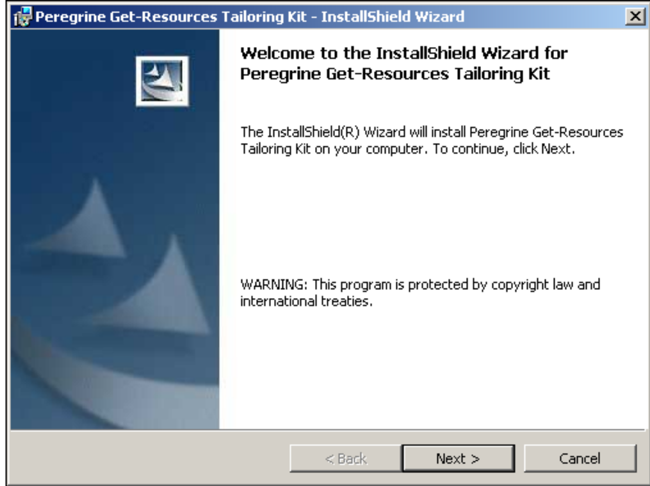
   The Get-Resources Tailoring Kit splash screen opens displaying a list of installation options.



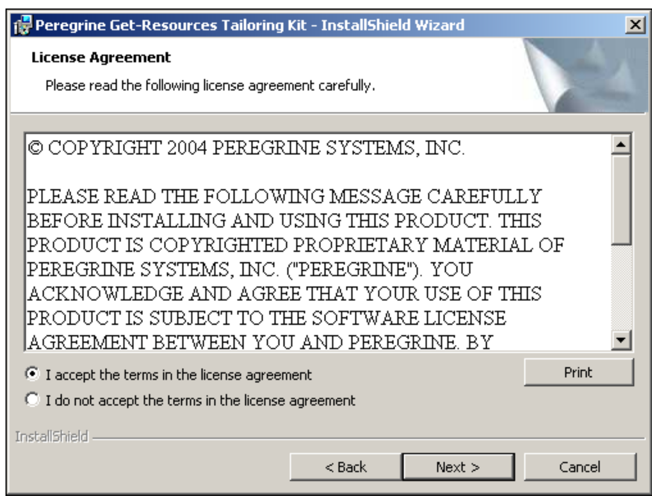
2 Install the required platform components for the Get-Resources Tailoring Kit.

Component	Action
Install Studio	Click this button to install Peregrine Studio 2.2.0.1068 on your system.
Install Get-Resources Tailoring Kit	Click this button to install Get-Resources Tailoring Kit 4.2 on your system.

Note: If you do not have Peregrine Studio, you must install it prior to installing the Get-Resources Tailoring Kit.

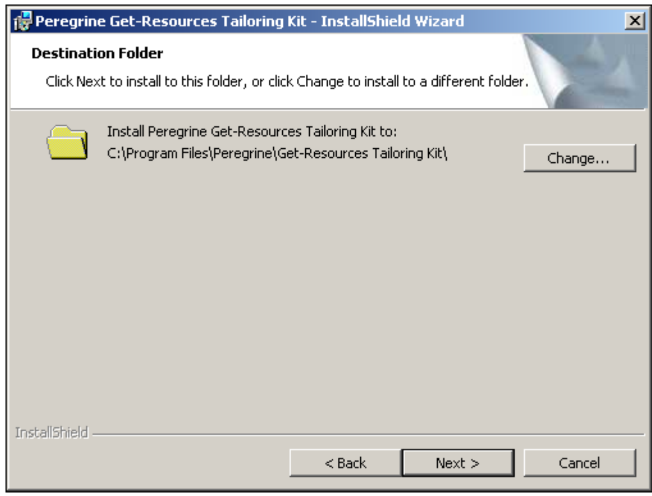


3 Click **Next** to read and accept the licensing agreement.

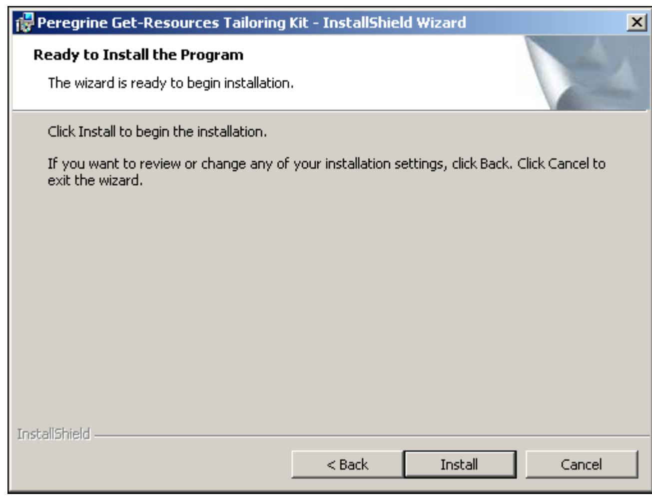


4 Select **I accept the terms in the license agreement**.

5 Click **Next** to select your destination folder.

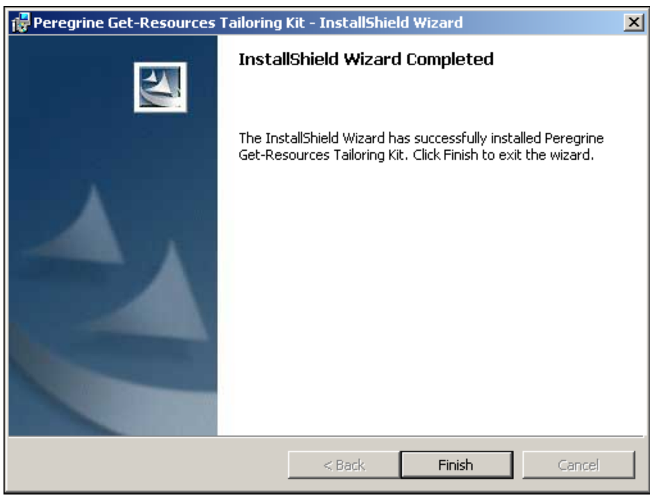


6 Click **Next** to accept the default installation location, or click **Change** to select another installation location, and then click **Next**.



7 Click **Install**.

After the installer copies and deploys the files to your system, the InstallShield Wizard Complete page opens.



8 Click **Finish** to close the InstallShield Wizard.

Upgrading the Get-Resources Tailoring Kit

During an upgrade installation, the customized changes are safe because studio stores the changes in a specific *package*, and all the corresponding files are stored in a folder that the upgrade does not touch.

After an upgrade, you can reload the customizations (patch package) into the new project, using the menu *File/Add package to project*.

The upgrade of tailored projects is always a challenge because the very same files that you patch with the tailoring kit are changed by the Peregrine product development team. Therefore, the following can occur:

Scenario	Description
1	If an element changed in the product between the two releases, and you patched it, the product change is not accounted for in the resulting build. If the change was either a bug fix or a new feature, you would not benefit from the change, even though the new version is installed. The patched element masks the product changes.
2	If an element is removed between two releases of the product, and you patched it, the result is a partial definition of the element that might not work.
3	If a group of elements changes between two releases of the product, and you patched only one element, the result might be an inconsistent group of elements that do not work properly.

As a result, there is always some work involved after upgrading a product to adapt the customization to the new product. To take care of the first scenario, you need to check what changed between two releases on every element that you patched, and merge in the changes you want to keep (bug fixes and new features). To solve the second scenario, you must verify if your patches still apply on an existing element. If not, the customization must be redone based on the new product.

Solving the third scenario is more complex. For every patch in your project, you need to assess if the changes are consistent with the way an element was patched. If the merge (as made by fixing scenario 1) is made properly, there should not be any problem.

Because of this difficulty in upgrading a customized system, Get-Resources 4.0 came with customization guidelines explaining how to modify the Get-Resources behaviors that are automatically upgradeable. For all changes made according to the guidelines, the steps to upgrade are much simpler:

1 Upgrade the product.

2 Upgrade the tailoring kit.

3 Open studio and re-add the customization package(s) to the newly installed project.

4 Re-build.

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 did not receive a Peregrine Studio authorization file, contact Peregrine Customer Support. You need this file to edit your Get-Resources files.

To open the Get-Resources project in Peregrine Studio:

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

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-Resources 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 from 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).
- J2SDK 1.3 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 J2SDK 1.3 or later.
- An installed instance of Get-Resources.
- A Web server.
- A Java-enabled application server.
- JavaScript-enabled Web browser (necessary to view changes to Get-Resources).

Note: See the latest compatibility matrix on the Peregrine support site for a list of supported Web servers, application servers, and 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 CHAPTER Using Studio

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:

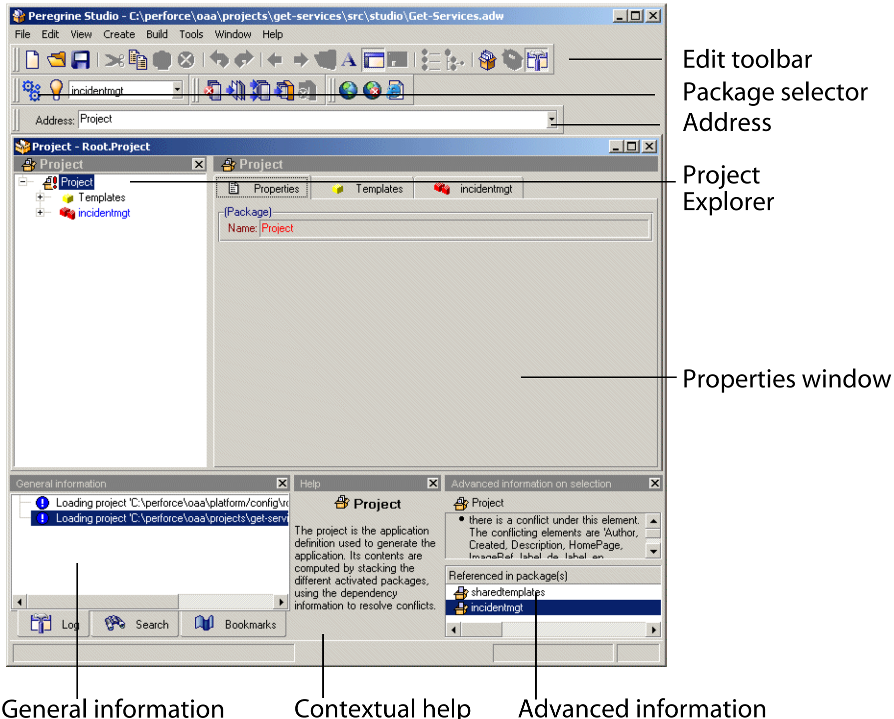
- The Peregrine Studio interface on page 27
- Best practices on page 32

The Peregrine Studio interface

The Peregrine Studio interface includes:

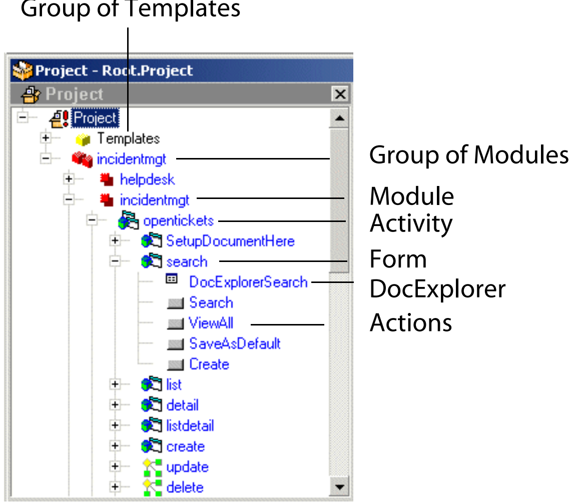
- Project Explorer
- Properties window
- Edit toolbar
- General information display
- Contextual help
- Address
- Package selector
- Advanced information

You can hide all elements of the interface except the Project Explorer and the Properties Window 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

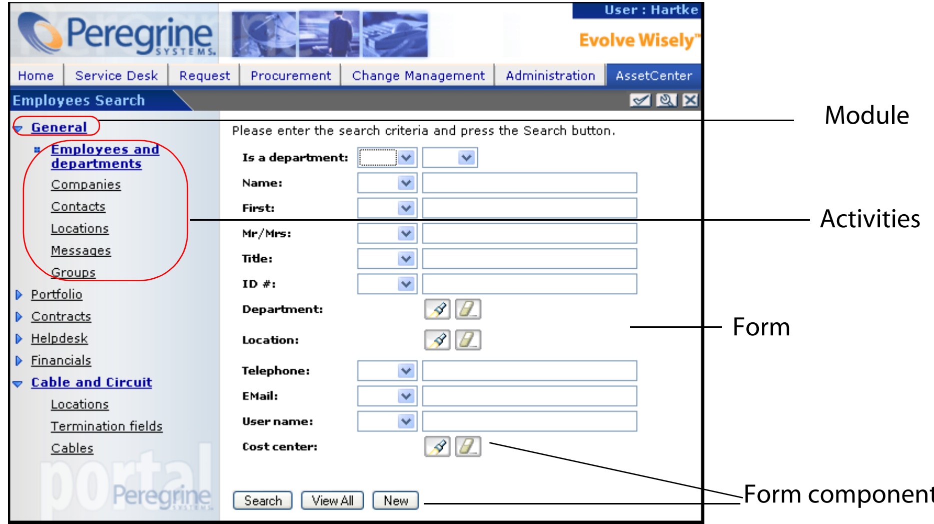
Right-click a node to display a list of context-sensitive options.

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.

Menu item	Description
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 Systems 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.

Do not change form definitions outside 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 (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 application server cache

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.

Use 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 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`).

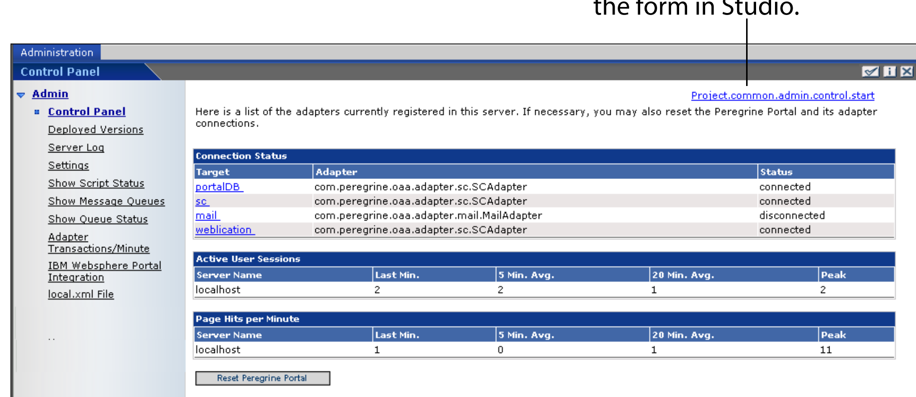
2 Click **Admin** > **Settings** to display the Settings form.

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 Peregrine Portal** 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 T**ools** > **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 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 (the orange arrow pointing to the left) to return to the component making the reference.

3 CHAPTER Peregrine Studio Projects and Packages

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 on page 39
- Building a project on page 45
- Peregrine Studio project packages on page 48
- Warnings for conflicts on page 52
- Deploying tailoring changes on page 53

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[1]	ECMAScripts create and format message objects to the Archway servlet. Get-Resources components will use ECMA message objects to display and process data.

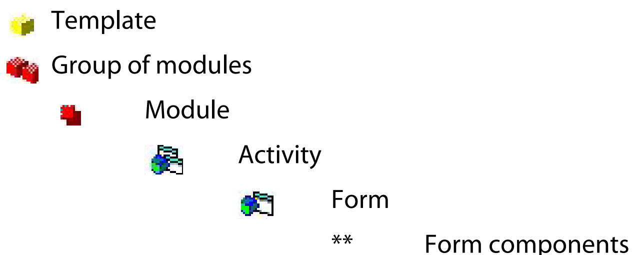
Studio component	Description
Document schema definitions	The XML files that define how the Archway servlet should format the ECMA message objects sent to and received from back-end databases. Get-Resources components will use the ECMA message objects to display and process data.
Presentation files	Any 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.

[1] ECMAScript is the core language standard shared between JavaScript and JScript.

For a listing of where Peregrine Studio saves and builds these files, see Project files on page 43.

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:



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.

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.
Group of modules	The group of modules component: Is the container for all the XML form definition 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).

Component	Description
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.
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.

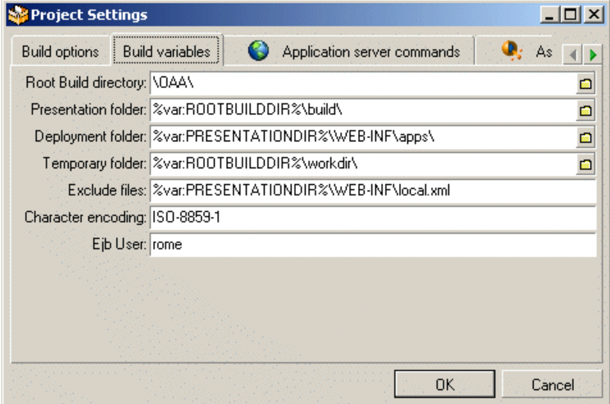
Warning: The Clean the target folders option cleans the folders listed in your build settings. If you build your project directly to your application server, then using this option deletes your installation of Get-Resources. It is recommended that you avoid using this option if you have installed Get-Resources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



2 Click the **Build Variables** tab.

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

Setting	Definition
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%`.

Presentation folder	This is the folder where your application is deployed. It can be:
	▪ the folder where your application server will look for the file to serve (recommended only for interim builds). For example:
	`C:\tomcat\webapps\oaa`
	▪ or the image folder that will be used to create the war file (recommended for final builds). For example:
	`C:\Program Files\Peregrine\Portal\image`
	Whatever path you enter here becomes the variable `%var:PRESENTATIONDIR%`.

Deployment folder	The folder where scripts, schemas, and XML form definitions are located. You do not need to change the value of this setting. Note: Do not change this option.
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.
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.
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.
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 definitions, 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.

Package type	Definition
System	The system packages provided by Peregrine Systems define the out-of-box functionality of Get-Resources.
Extension	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.

Term	Definition
Name	Enter a name for the new package extension. The package extension name cannot contain spaces or special characters.
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 50 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 are saved in your new package.

Activating and deactivating packages

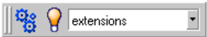
You can control the packages and package extensions that are part of Get-Resources by activating or deactivating them from the Package Activation menu. To include a package in Get-Resources installation, activate the package, and then build the Studio project. To remove a custom package from your installation,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



2 Click the Package activation button.

3 Select the check box 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: Deactivating 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.

4 CHAPTER

Studio Components

This chapter contains a list and description of all of the components you can add to a Project in Studio. The information follows the menu structure of these components and subcomponents as you see them in Studio.

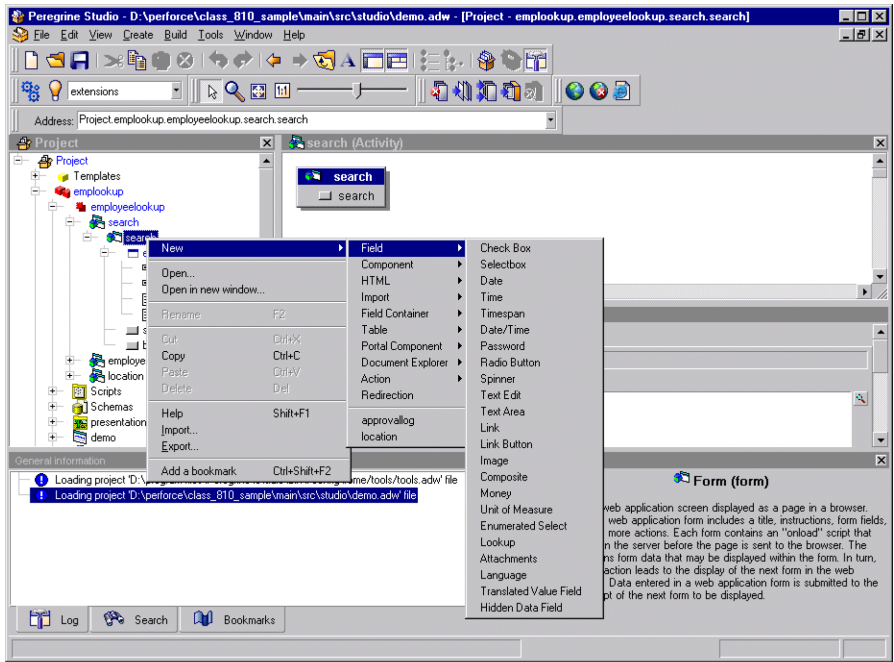
The menus that you see when you open the Get-Resources package in Peregrine Studio may vary slightly from the menu options documented here. Menu options change depending on the components you created. For example, you must have the folder called `shared templates` in your package to enable DocExplorer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

This chapter covers the following topics:

- Adding components on page 56
- Types of form components on page 67

Adding components

To add components to your Project, right-click on the node to which you want to add a component. A menu of options opens.



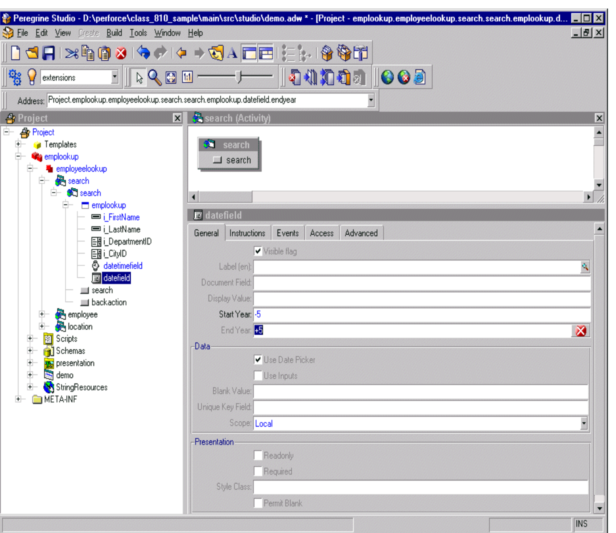
Project > New >

Component	Description
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 control access rights. From the Group of Modules, you can create the following.

Component	Description
Module	Get-Resources contains modules. The role that a user takes in performing tasks often determines the modules. For example, you can design one module for employees who open requests for service. Another module can be for managers approving requests. Modules typically have specific access role restrictions so that only those users who need to perform the module's task have permission to do so.
Peregrine Portal Activity	Each module contains one or more activities that define the steps users can take to complete the module's task. For example, a Request module can have activities for browsing catalogs, reviewing a shopping cart, and filling out a request form. Each activity typically displays in Get-Resources on a sidebar menu at the left of a form. Activities typically have 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 goes to the browser. The script obtains form data to display 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.

Component		Description
Field >		
	Check Box	Allows you to toggle a value on or off.
	Selectbox	Allows you to select a value from a list displayed in a Combo Box field.
	Date	Allows you to view or enter a date. You can enable or disable (the default is *enabled*) an optional calendar form component (Date Picker) 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.
		
	Time	Allows you to view or set a time value.
	Timespan	Allows you to view or edit a timespan value.
	Date/Time	Allows you to view or set a date and time value. There is an optional calendar form component (Date Picker) that you can enable or disable in Studio (the default is *enabled*). See Date component.
	Password	Allows you to enter a password.
	Radio Button	Allows you to select one of several choices using radio buttons.
	Spinner	Allows you to enter a numerical value. The control allows you to type the number in directly. It also allows you to select a number by clicking on the spinner buttons that increase and decrease the value.

Component	Description	
Field> (continued)	Text Edit	Allows you to display or edit a value in a plain text field.
	Text Area	Allows you to enter text into a multiline edit field.
	Link	Displays a hyperlink that the user can click 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 you to view or edit a monetary value.
	Unit of Measure	Allows you to view or edit a value that is a unit of measure.
	Enumerated Select	Allows you to select a value from a list displayed in a Combo Box field.
	Lookup	Allows you to enter a value by performing a lookup operation. The lookup is done in a separate pop-up window.
	Attachments	Allows you to view and add attachments to a document.
	Language	Allows you to select a 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 you 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.

Component	Description	
HTML >		
	Blank Line	Adds a blank vertical line to the form.
	Free-form HTML	Allows you to insert arbitrary HTML into a form as well as insert client-side JavaScript into a Web page. Note: Peregrine Systems recommends that you move large amounts of client-side JavaScript to a presentation file that the page can then import.
Import >		
	Static Import	Imports the text content of a file for inclusion in a Web page. You can import files that define static HTML, JSP code or client-side JavaScript functions.
	External HTML Plug-in	Includes dynamic content into the form. At run time, the URL that the plug-in references is accessed by the server, returning contents which then insert 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. You can divide fields 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. Peregrine Systems recommends 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.

Component	Description	
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 Configurative	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.

Component	Description	
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 redirects according to the condition that the script returns. 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 that the Web application forms uses.	
	Header	Initial comments and imports required in this script file.
	Function	Script function defining application logic executed on the server. Make sure that all functions that have public access 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 you use it as an onload script for a form or if you call it directly using an Archway HTTP message.
Group of Scripts	Server-side ECMAScripts.	

Component	Description	
Group of Triggers	A collection of triggers. Get-Resources does not use this container.	
Trigger	Individual trigger for a document. Get-Resources does not use this component.	
	Message action	Message action that the trigger executes.
	Workflow action	Workflow action that the trigger executes.
	Script action	Script action that the trigger executes.
	Bizdoc Java action	Java action that the trigger executes inside Bizdoc.
Group of Triggers	Collection of triggers. Get-Resources does not use this component.	
	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 used in your Web application.	
Image	The image loads into the ImageData property as binary data. The system uses the file name property only the first time to load the image.	
Group of Images	Folder containing image files.	
	Image	The image loads into the ImageData property as binary data. The system uses the file name property 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	Any generic file in the Presentation folder that the Web server needs; for example, client-side JavaScript, static JSP files.	
Binary	Binary file outputted in the presentation folder. Accessed by the Web server and used by the browser.	

Component	Description	
Group	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 the Web server needs; 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 that the user can add to the home page.	
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 that a Web application uses. 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 >

Component	Description
Style Sheet	Not supported.

Group of Roles

Component	Description
Group of Roles	Not supported.

Group of Files > New >

Component	Description
String file	Temporary representation of a string file.
Ini file	Temporary representation of a `scriptpoller.ini` file.

Group of Strings > New >

Component	Description
Multilingual string	The name of the StringResource is the ID of the string.

Entities (collection of business objects) > New >

Component	Description
Entity	Get-Resources does not use this component.

Interfaces > New >

Component	Description
Interface	Not supported.

System enumerations > New >

Component	Description
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	Define one value for a system enumeration.

Templates > New

Component	Description
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	Get-Resources does not use this component.
Workflow action	Get-Resources does not use this component.
Bizdoc Java action	Get-Resources does not use this component.
Script action	Get-Resources does not use this component.
Trigger	Get-Resources does not use this component.
Group of Images	Allows you to create a group of images.
Attribute	Get-Resources does not use this component.
Reference	Get-Resources does not use this component.
Contain	Contains an object as an embedded member.
Computed	Computed property.
Structure	Get-Resources does not use this component.
Collection	Get-Resources does not use this component.
Methods	Get-Resources does not use this component.
Entity	Get-Resources does not use this component.

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 that stores 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 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 can 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 using 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 the following attributes for a component template.

Title	User Role Restrictions
Summary	Dynamic Runtime Restrictions
Order	

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 the following form components to a fieldsection.

Field	Header
Component	Import
HTML	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 the following attributes for a fieldsection.

Title	Dynamic Runtime Restrictions
Summary	Border
Order	Readonly
User Role Restrictions	

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 the following form components to a text edit field.

None

Attributes

You can set the following attributes for a text edit field.

Instructions	Display Value
Label	Max Characters
Title	Data
Document Field	

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.

Option	Value
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.

- 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.
- 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.

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 the following form components to a selectbox field.

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

Attribute categories

You can set the following attribute categories for a selectbox field.

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

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.

Attribute	Description
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 passes 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 the following form components to a hidden data field.

None

Attributes

You can set the following attributes for a hidden data field:

Document Field	Unique Key Field
Display Value	User Role Restrictions
Visible Flag	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 the following form components to a redirection.

None

Attribute categories

You can set the following attribute categories for a redirection.

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

Redirection attributes

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

Attribute	Description
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* in this guide 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.
- Describes how the data will be displayed in the columns of the table.

A simple table requires columns components 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 the following form components to a simple table.

Link	Spinner Column	Image Column
Text Column	Radio Button Column	Link Column
Entry Column	Checkbox Column	Lookup Column

Attributes

You can set the following attributes for a simple table.

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

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 generates the table data.
- Describes how the data displays 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 the following form components to a document table.

Column

Attributes

You can set the following attributes for a simple table.

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

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 you 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 typically have 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 the following form components to a table link.

None

Attributes

You can set the following attributes for a simple table.

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

Table link attributes

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

Attribute	Description
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 the following form components to a text column.

None

Attributes

You can set the following attributes for a text column.

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

Attribute	Description
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 the following form components to a form column.

Any

Attributes

You can set the following attributes for a form column.

Visible Flag	User Role Restrictions
Label (en)	Dynamic Runtime Restrictions
Order	

Form column attributes

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

Attribute	Description
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	Description
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.

Action	Description
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 the following form components to an action.

None

Attributes

You can set the following attributes 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
Presentation

Action attributes

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

Attribute	Description
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 CHAPTER Scripting

This chapter provides an overview of how to use scripts. You must be familiar with JavaScript and ECMAScript and have access to the JavaDocs provided with your Get-Resources Tailoring Kit installation CD.

This chapter covers the following topics:

- Overview of scripts on page 83
- Testing scripts on page 92
- Common message operations on page 97
- Using ECMAScript in an object-oriented manner on page 100
- Sample scripts on page 105
- References on page 113

Overview of scripts

Get-Resources uses scripts to query back-end databases and to format the results into XML documents based on schemas. Generally, you 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. Before creating or making changes to a script, you must create or activate 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.

Script types

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

Scripting type	Description
Server-side	Server-side scripts run from a Web server. Server-side scripts have access to both user-submitted form data and any data that a back-end system generates. You can return the output of server-side scripts to both a back-end system and the remote browser. All Get-Resources server-side scripts use ECMAScript. An example of server-side scripting is querying a back-end system for the list of items associated with a particular order.
Client-side	Client-side scripting runs from a JavaScript-capable browser. Client-side scripts have access to user data before submitting to a Web server and any back-end data that uploads with the current Web page. Only the client browser uses the output of client-side scripts. All Get-Resources client-side scripts are in JavaScript. An example of using client-side scripting is updating the total price displayed on an order form when a user enters an amount in another field of the page.

Where to store scripts

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

Script type	Language	Where created and stored
Server-side	ECMAScript	You can author server-side scripts in Peregrine Studio. Each script then becomes an object available for use throughout the project. It is possible to create scripts outside of Peregrine Studio that can then be used in schema extensions, or as a Get-Resources script interface. You can also create scripts as script extensions to modify the behavior of existing scripts (see the Get-Resources Admininstration Guide).
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 forms. At run time,

the deployment application server executes the forms along with any server-side scripts that the forms call and sends the output to the client browser. The client browser executes any client-side JavaScript present in the rendered HTML page.

How to use scripts

The Archway servlet supports several different methods to invoke and utilize scripts within Get-Resources. The following sections describe the different ways to use ECMAScript and JavaScript 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.

This API	Sends the following
sendDocQuery	SQL or XML document query to the back-end database. Archway queries the record using the table and field information that the schema supplies. The database then returns the results of the query as an XML document formatted as defined in the schema.
sendDocInsert	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 that the schema supplies.
sendDocUpdate	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 that the schema supplies.
sendDocDelete	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 that the schema supplies.

Get-Resources typically uses 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);
```

Parameter	Value
adapter name	Enter the name for the back-end database adapter. This adapter uses the ODBC connection that you defined in the `archway.ini` file. For most applications, the adapter is a two-letter name.
schema name	Enter the name defined in the `<document name="schema name">` element of the schema file.
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 defined in all onload functions. The input message is the XML message containing the HTML page parameters.
message object	Enter a variable name of a message object containing a schema name and any input parameters.

For example, the following script sample 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 Systems does not provide customer support for custom client-side scripts.

Editing an existing script

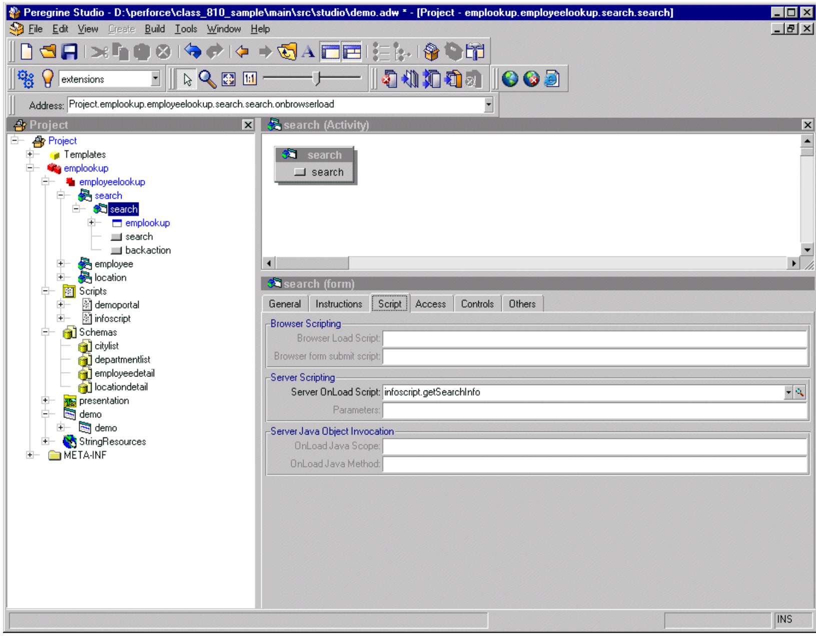
You can edit the ECMAScript in your project directly from the Peregrine Studio interface.

Important: You may lose changes that you make to existing scripts when you next upgrade. Use script extensions whenever possible to avoid this problem.

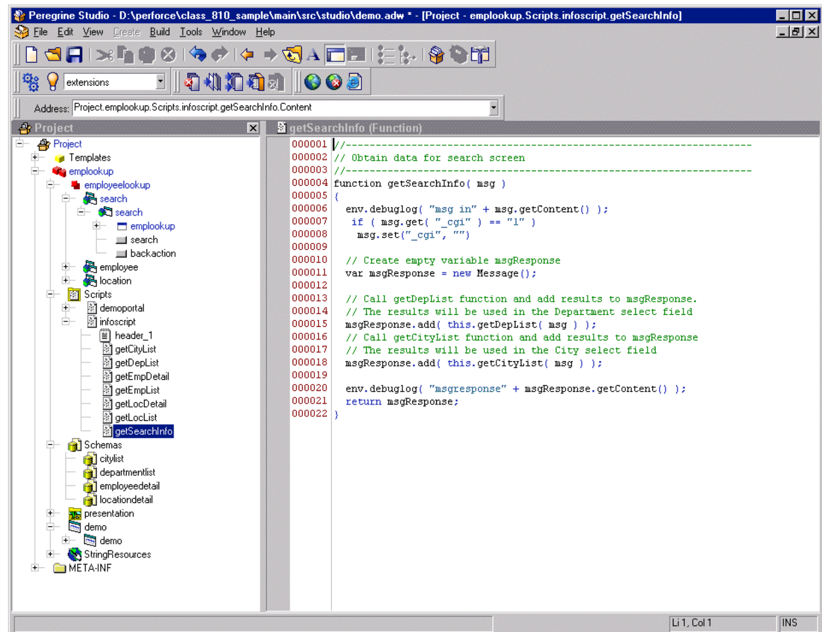
To edit an existing script:

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.

The script update loads 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 script types.

Script type	Description
Form onload script	Run these scripts to gather data for non-DocExplorer forms. Peregrine Studio stores form onload scripts below the first Group of Scripts node. These typically are **Scripts** or **ServerScripts**.
Preexplorer	Run these scripts to manipulate the XML document that renders in the Get-Resources interface. Peregrine Studio stores preexplorer scripts below the Preexplorer Group of Scripts node.
Preload	Run these scripts to gather data for DocExplorer forms. Peregrine Studio stores preload scripts below the Preload Group of Scripts node.
Schema	Run these scripts before or after an adapter connects with the back-end database. Peregrine Studio stores schema scripts below 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 below the Group of Scripts.

3 Type 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 below the Script node.

5 Using the text editor window, type 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 below the Script node.

7 Using the text editor window, type 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 loads into Get-Resources.

Date values in scripts

In server-side scripts, all dates in the XML messages must use the internal format `YYYY-MM-DD`. The format for timestamps is `YYYY-MM-DDTHH:mm:SS.SSSZ`, where

Symbol	Value
T	Is the letter T.
HH	Specifies the hours in 24 hour format.
mm	Specifies the minutes.
SS.SSS	Specifies the number of seconds and milliseconds.
and Z	Indicates the time zone.

The following is an example of the format showing GMT:

```
2004-02-19T16:58:23+00:00
```

The following is an example of the format using the name of the time zone:

```
2004-03-29T07:00:00America/Los_Angeles
```

Note: The names of time zones are in the `Java.util.TimeZone class`.

Timestamps are usually in the GMT time zone. However, dates on server machines do not need to be in this format. The user interface automatically converts dates to and from the local date format automatically whenever the system uses date widgets or date columns.

When you set the date manually in an XML message, you may need to manipulate its format. Use the `DataFormatter.getArchwayDate` and `DataFormatter.getArchwayDateTime` functions.

See Working with dates in scripts on page 112 for examples of date manipulation.

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 Mozilla provides. The Rhino JavaScript Debugger provides a graphical user interface for debugging interpreted 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 and configure your application server to run as a service, you must set up your service to interact with the desktop.

To enable the Rhino JavaScript debugger:

1. Log on 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. Log on to the Get-Resources server.

6 Browse to the Get-Resources deployment directory. By default, the directory path is:

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

Parameter	Value
<application server>	Enter the installation path to your application server. For example, `C:\Program Files\Peregrine\Common\Tomcat4`
<context>	Enter the path where you deployed the Get-Resources files. For example, `webapps\oaa`.

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

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

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

9 Save the file.

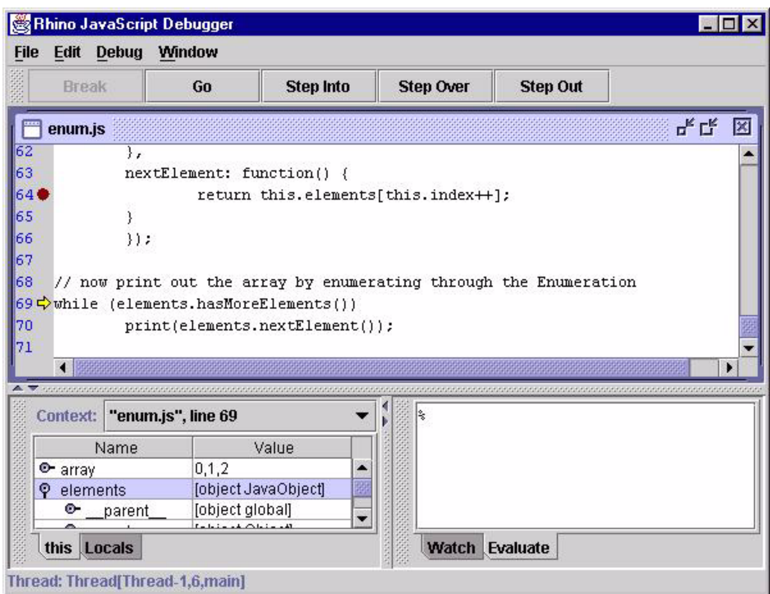
10 Copy the `rhinodebugger.jar` file from the Get-Resources Tailoring Kit Installation CD to the following path on your test server:

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

Parameter	Value
<application server>	Enter the installation path to your application server. For example, `C:\Program Files\Peregrine\Common\Tomcat4`
<context>	Enter the path where you deployed Get-Resources the files. For example, `webapps\oaa`.

11 Restart your application server.

The Rhino JavaScript Debugger opens the next time you start your application server on this system.



For more information about the Rhino JavaScript Debugger, go to the Mozilla Web site: *http://www.mozilla.org/rhino/debugger.html*.

URL queries

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

Archway invokes the server script or schema as an administrative user and returns the output as an XML document. Your browser must have 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 on to 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
```

Parameter	Value
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.
oaa/servlet	This mapping assumes that you are using the default URL mapping that Get-Resources automatically defines for the Archway servlet. If you defined another URL mapping, replace the servlet mapping with the appropriate mapping name.
script name	Enter the name of the script you want to run.
function name	Enter the name of the function that the script uses.

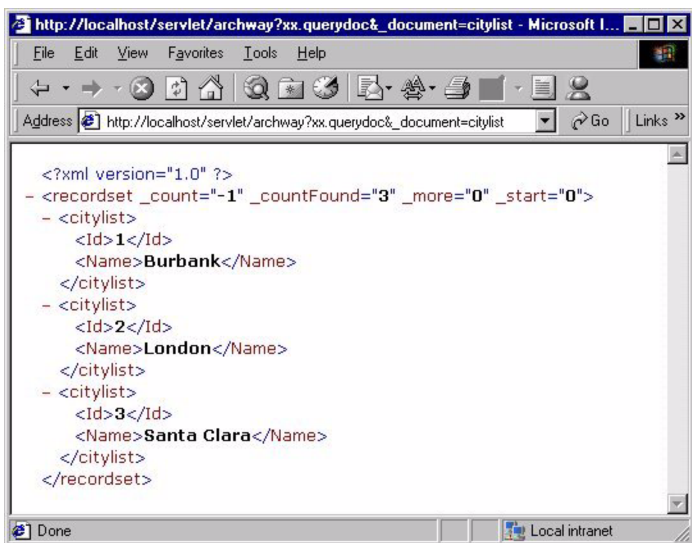
URL schema queries

Archway URL schema queries use the following format:

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

Parameter	Value
oaa/servlet	This mapping assumes that you are using the default URL mapping that Get-Resources automatically defines for the Archway servlet. If you defined another URL mapping, replace the servlet mapping with the appropriate mapping name.
adapter name	Enter the name for the back-end database adapter the schema uses. The adapter uses the ODBC connection that you defined in the Admin module Settings page.
schema name	Enter the name defined in the `<document name="schema name">` element of the schema file.

Make sure that your script output resembles the following example.



URL SQL queries

Archway URL SQL queries use the following format:

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

Parameter	Value
oaa/servlet	This mapping assumes that you are using the default URL mapping that Get-Resources automatically defines for the Archway servlet. If you defined another URL mapping, replace the servlet mapping with the appropriate mapping name.
adapter name	Enter the name for the back-end database adapter the schema uses. This adapter uses the ODBC connection that you defined in the Admin module Settings page.
table name	Enter the SQL name of the table you want to query from the back-end database.
field name	Enter the SQL name of the field you want to query from the back-end database.

Parameter	Value
value	Enter the value you want the field or optional parameter to have.
_[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.

Common message operations

The following section describes some common methods that server-side scripts use to create XML messages. Refer to the JavaDocs, especially `com.peregrine.oaa.core.Message`, for more information and examples about 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();
  ```

  This 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" );
  ```

  This 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" );
  ```

  This 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" );
  ```

  This 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" );
  ```

  This 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" );
  ```

  This 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 to activate Peregrine Studio form components such as redirections and access fields.

- 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 ...
}
```

This 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 92.

Using a logging domain, you can 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 because script logging is CPU-intensive and degrades server performance.

Using ECMAScript in an object-oriented manner

ECMAScript implementation in Get-Resources

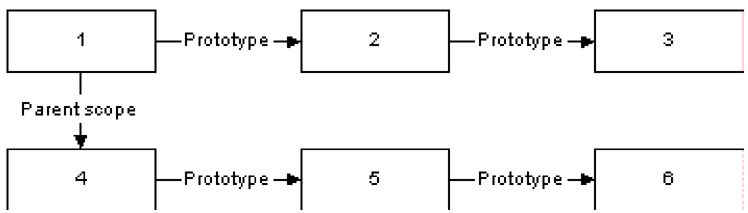
All Scripts defined in Peregrine Studio load as one ECMA script object. The functions defined in the Script are the object's methods, and the variables declared outside a function are the object's attributes. This implementation as an object is what enables you to use the dot syntax to call scripts and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

Name resolution in ECMAScript

Every ECMAScript object has a special property: its *prototype*. A prototype is an ECMA script object 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 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
```

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

Overwriting a method to extend the parent class method is 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 prints shows 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, the parent function must be called in a specific way, as shown 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
```

This delivers the expected result. The code is using the apply method of the Function object, passes the object 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 seems 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__` is evaluated against the derived object, and the start function in `sportscar2` continues to call itself. It is therefore preferable to store the parent object in a variable that the subclasses do not overwrite. Here, with a nomenclature that uses the parent prefix and the parent class name, the uniqueness is ensured. Try this with a racecar

class that derives from `sportscar2` and overwrite the start function by calling the parent.

How to use object orientation for tailoring

In Get-Resources, objects instantiate automatically from the script files when they load in memory. To implement the prototype hierarchy, you must set the `__proto__` attribute 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, see the list of suggested reference materials on page 113.

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 to 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 can 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 following XML output is an example of the kind of data using a similar script returns.

```
<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 that your schema query returns.

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 can 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

following XML output is an example of the kind of data that using a similar script returns.


```
<_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 examples 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 includes a subdocument, `Department`, in addition to any other input data passed to the script. This subdocument searches 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 the input message of a search 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 following XML output is an example of the kind of data that using a similar script returns.

```
<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 that the search form generates. 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 that the search results return. 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 that the query generates 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.

Working with dates in scripts

The following code samples demonstrate tasks related to date manipulation.

To get the string that corresponds to the current date:

```
// Gets current date
var date = new Date();
// Gets the current date and time string
var strDateTime =
DataFormatter.getArchwayDateTime(date.getTime());
// Get the current date string
var strDate = DataFormatter.getArchwayDate(date.getTime() -
date.getTimezoneOffset()*60000);
```

To get a date value from the internal OAA format:

```
var strDAssignment = msg.get("dAssignment");
var lMsAssignment =
DataFormatter.getDateTimeInMilliseconds(strDAssignment);
```

To get a date and time value from the internal OAA format:

```
var strDtInvent = msg.get("dtInvent");
var lMsInvent     =
DataFormatter.getDateTimeInMilliseconds(strDtInvent);
```

Note that these numeric values are very convenient for comparing dates, performing arithmetic calculations on dates (such as calculating a duration and adding an amount of time to a date), and other tasks. From these numeric values, you can get an ECMAScript Date object.

```
var dateAssignment = new Date(lMsAssignment);
var dateInvent = new Date(lMsInvent);
```

In addition, you can get a Java date object.

```
var jdateAssignment = new
Packages.java.util.Date(lMsAssignment);
var jdateInvent = new Packages.java.util.Date(lMsInvent);
```

To display a date value in a user-friendly format:

```
var strUserDtAssignment = user.getUserFormat(strDAssignment,
"date", null);
var strUserDtInvent = user.getUserFormat(strDtInvent,
"datetime", null);
```

To get the internal OAA format for a date and time:

```
var strOAADtInvent1 =
DataFormatter.getArchwayDateTime(lMsInvent);
var strOAADtInvent2 =
DataFormatter.getArchwayDateTime(dateInvent.getTime());
var strOAADtInvent3 =
DataFormatter.getArchwayDateTime(jdateInvent.getTime());
```

To get the internal OAA format for a date only:

```
var strOAADAssignment1 =
DataFormatter.getArchwayDate(lMsAssignment);
var strOAADAssignment2 =
DataFormatter.getArchwayDate(dateAssignment.getTime());
var strOAADAssignment3 =
DataFormatter.getArchwayDate(jdateAssignment.getTime());
```

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 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.

CHAPTER 6

Tailoring Tasks

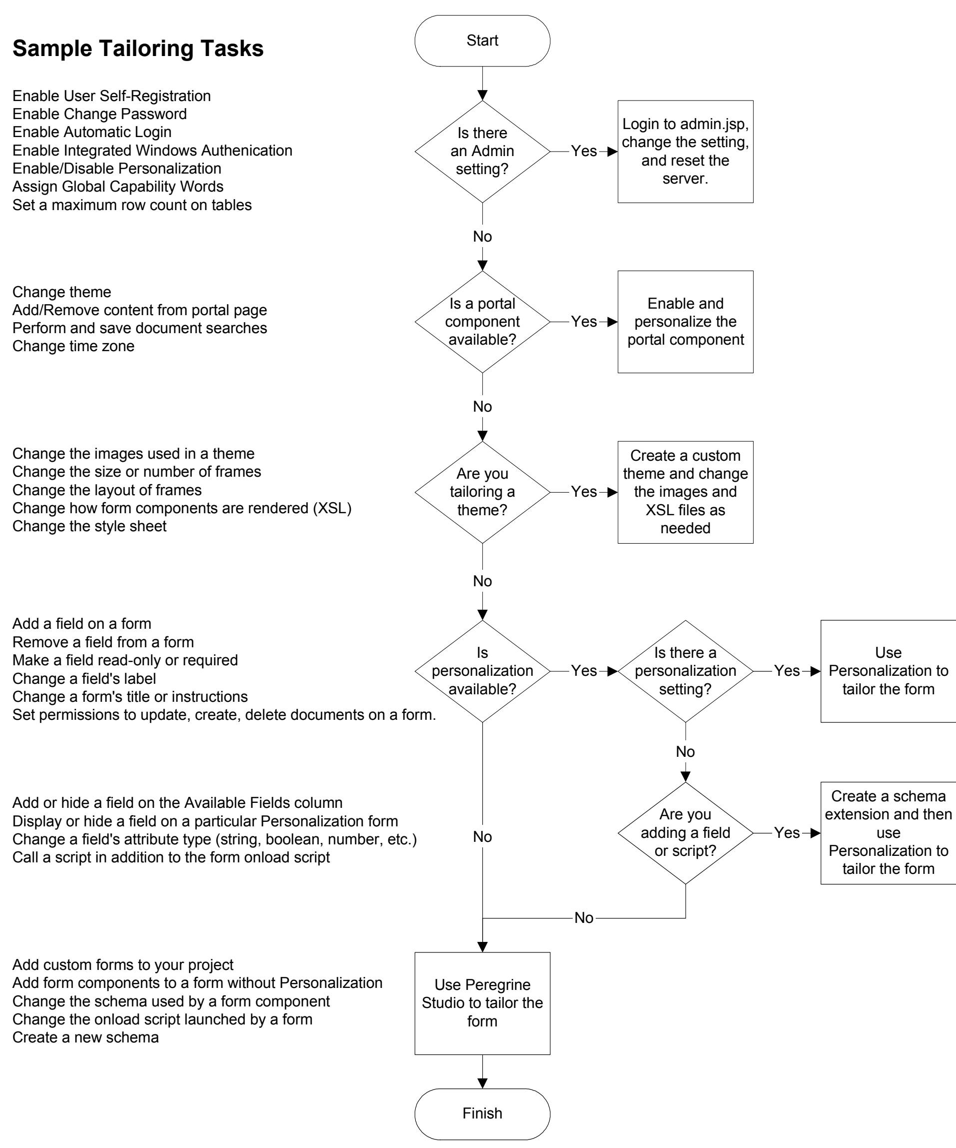
The following chapter lists all the tailoring tasks you can perform with the Get-Resources tailoring kit.

This chapter covers the following topics:

- Tailoring workflow on page 116
- List of tailoring tasks on page 117
- Tailoring forms and components on page 119
- Tailoring Get-Resources forms on page 138
- Adding personalization on page 152
- Tailoring scripts on page 159
- Creating custom schemas on page 178
- Adding data validation on page 187
- Assigning default values on page 190
- Translating tailored modules on page 195

Tailoring workflow

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



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:

- Changing a form's title on page 120
- Changing a form's instructions on page 121
- Changing a form's onload script on page 122
- Changing a form component's label on page 123
- Hiding a form component on page 124
- Changing a form component to read-only on page 125
- Changing the schema that a form component uses on page 126
- Changing the document field that a form component uses on page 127
- Displaying a form within a frameset on page 130
- Adding Get-Resources to an existing frameset on page 132
- Displaying a script variable in a form component on page 133
- Creating a portal component on page 134
- Tailoring Get-Resources forms on page 138

  Best Practices on page 138

  Changing the request summary screen on page 139

  Changing the catalog select list on page 144

  Changing the purchase order summary screen on page 146

  Changing the purchase order line detail screen on page 149

  Changing the request line selection list on page 151

DocExplorers

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

- Adding personalization on page 152
- Adding a DocExplorer reference on page 153
- Personalizing a DocExplorer reference on page 155
- Adding personalization form components – lookup fields on page 156

Scripting

You can use the following scripting methods for tailoring:

- Editing an existing script on page 159
- Adding a custom script on page 162
- Changing request behavior on page 163
- Example: adding a field from one schema to another schema on page 165
- Changing purchase order behavior on page 168

Schemas

You can tailor schemas in the following ways:

- Adding logical and physical mappings to your schema on page 180
- Adding a schema to your Peregrine Studio project on page 179

Data validation

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

- Adding data validation on page 187
- Making a field required on page 187
- Setting request line default values from catalog entries on page 170
- Purchase order validation on page 190
- Purchase order line default values on page 194
- Request validation on page 188
- Purchase order validation on page 190

Default values

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

- Setting request default values on page 190
- Setting request default values on page 190
- Request line default values on page 170
- Purchase order default values on page 194
- Setting request line default values from catalog entries on page 170
- Setting request line default values to values in a request on page 192
- Purchase order default values on page 194
- Purchase order line default values on page 194

Translation

You can translate your tailored forms in the follow ways.

- Editing existing translation strings files on page 195
- Adding new translation strings files on page 198

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.

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.

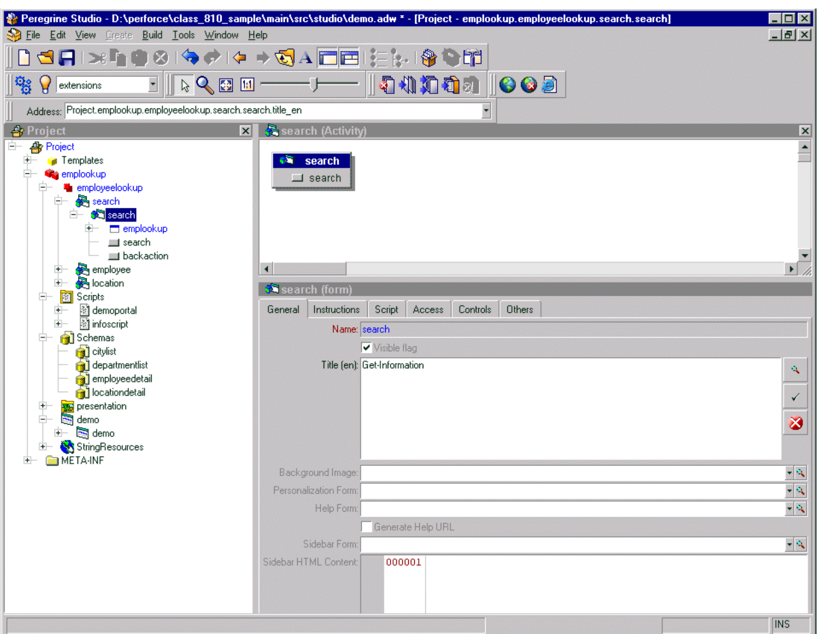
To change a form title

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:

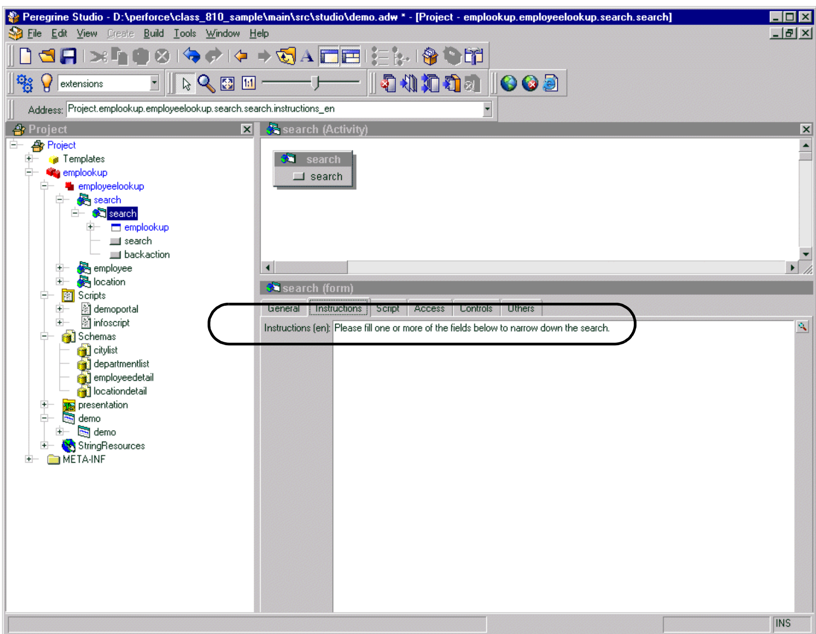
1 Select the form in the Project Explorer.

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 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 that a form invokes:

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).

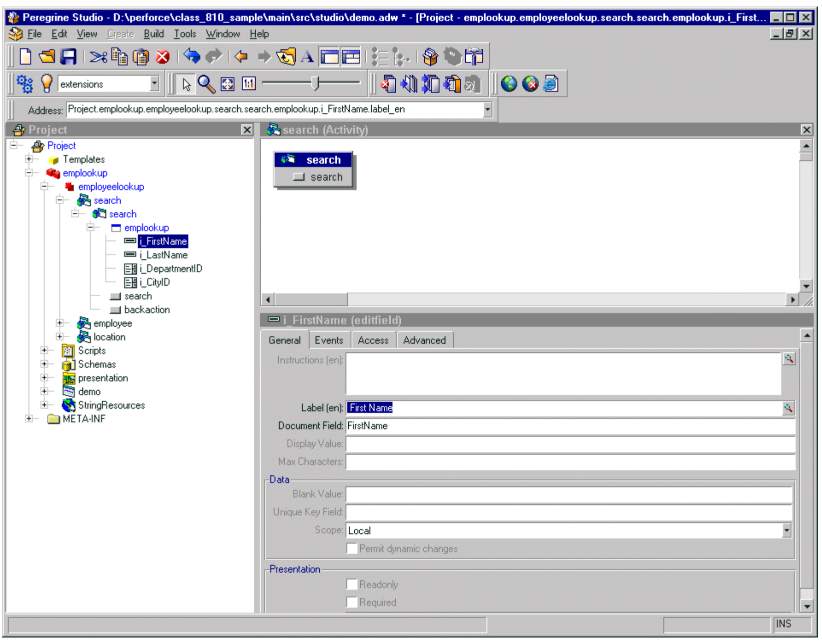
To change a component label (field label):

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.

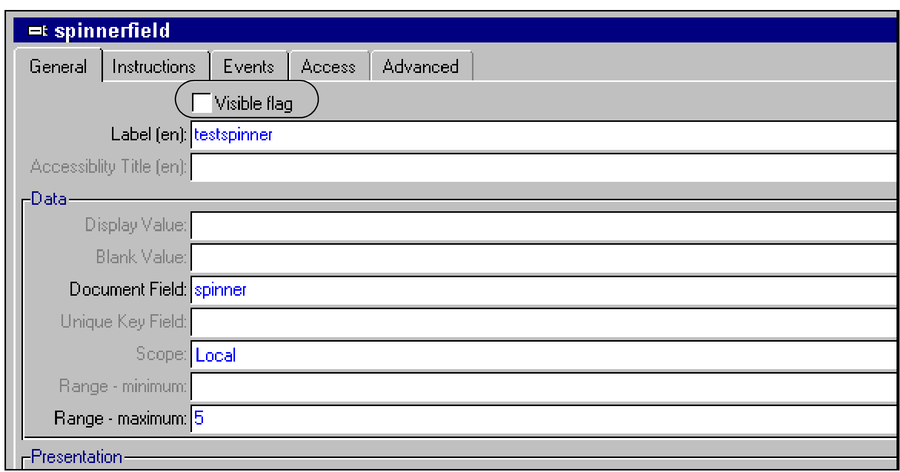
To hide a form component in the interface:

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.



Warning: If you clear the check box beside the Visible flag option, that component is not included in the generated HTML.

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 read-only attribute for the form component. The data displayed by a read-only 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 read-only attribute.

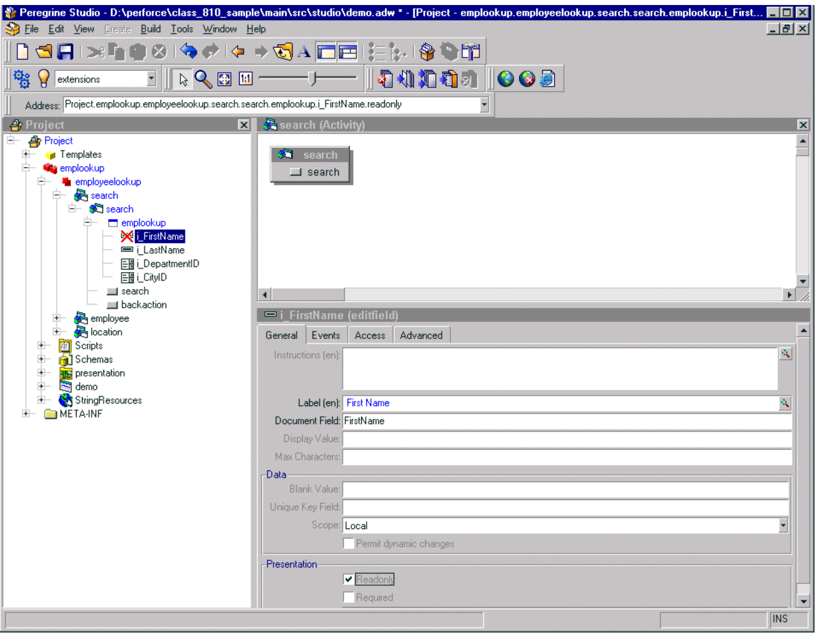
To make a form component read-only:

1 Select the form in the Project Explorer.

2 On the General tab, select the **Read-only** check box.

3 Save your project.

4 Build your project file.



Changing the schema that a form component uses

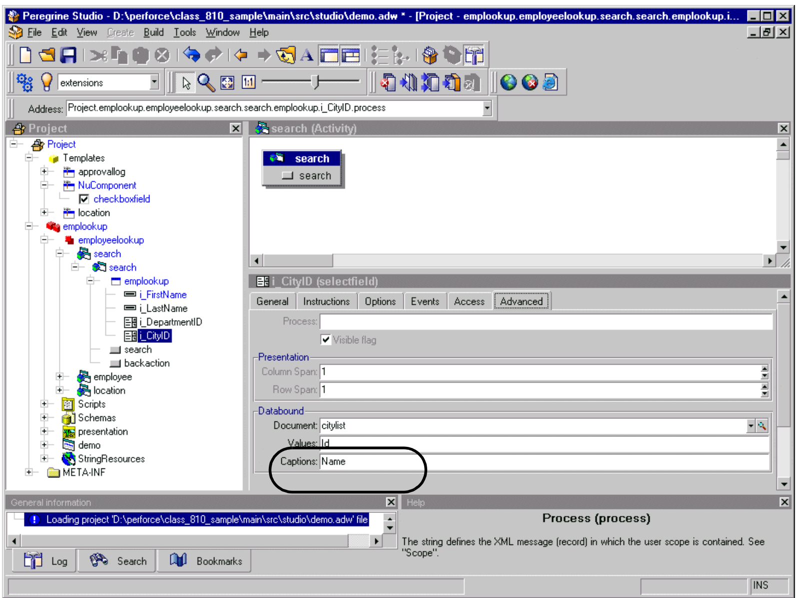
Certain form components such as selectfields and simple tables use a schema to determine what information to display. You can change the information these form components display by changing the schema defining the document fields. In some cases you may also need to change other form component attributes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

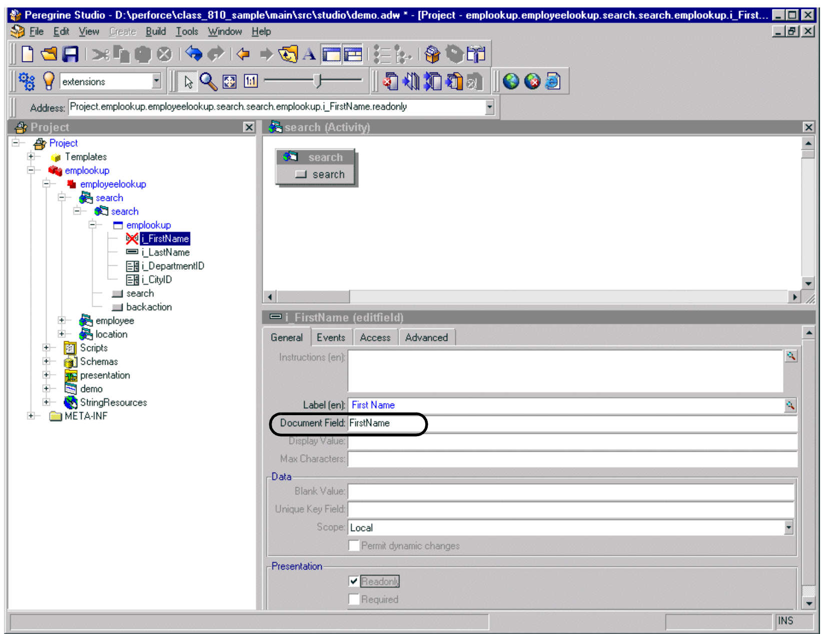
1 Select the form component in Peregrine Studio to display the component's properties.

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 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. You can format a typical asset request 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.

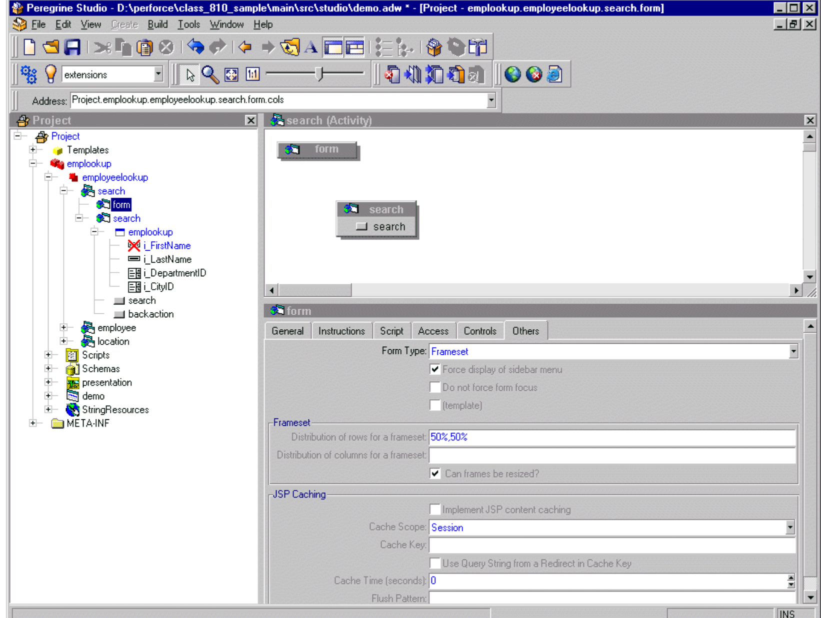
Displaying a form within a frameset

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.

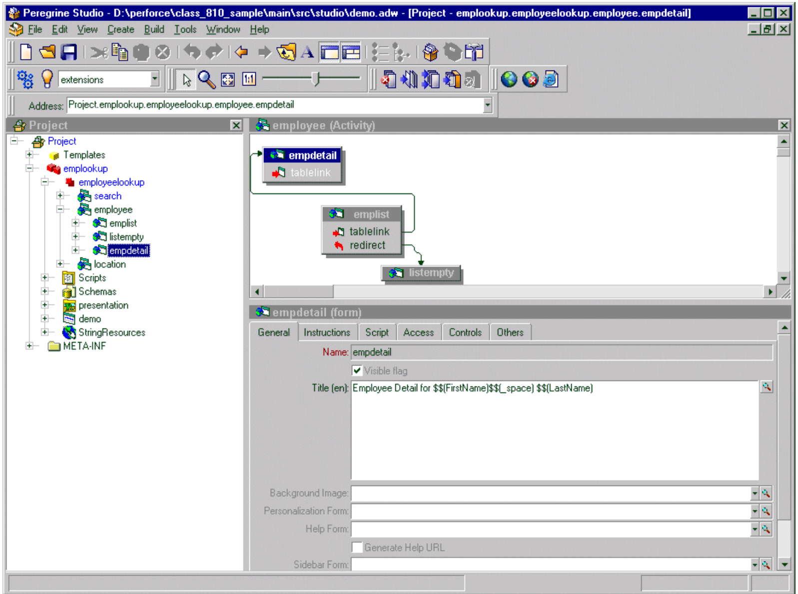
```
<script language="JavaScript" SRC="js/setDomain.js">
</script>
```

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:

Property	Description
Label (en)	Enter the name you want the portal component to have in the Add/Remove content page.
Column type	Select either wide or narrow. This setting determines the size of the portal frame where Get-Resources displays the portal component.
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.

Use the following format for the URL:

e_moduleName_activtyName_formName.do?property=value

Parameter	Description
For `moduleName`	Enter the module where your configuration form resides.
For `activityName`	Enter the activity 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.

Setting	Description

`Server OnLoad Script`	This script must either be set to or call the `portal.editComponent` script.
	Note: 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);`
	Your custom script must preserve the value of the `_Id` variable that the **portal.editComponent** function passes to it.

`Save action`	You must add a save action to your configuration form that has a target-form of **portal.edit.save**.
No sidebar navigation	All of the out-of-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.
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 of changes or revisions. This usually is a hidden field that is not 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

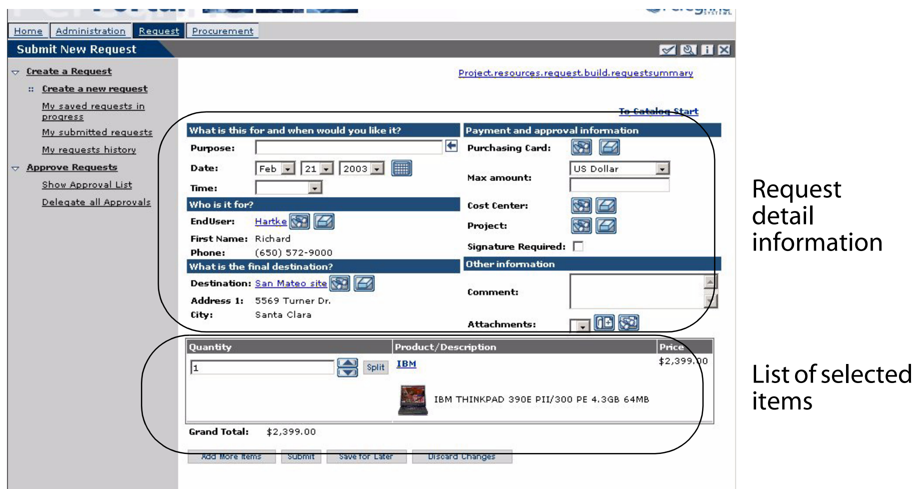
The following general tips will enhance the ability to upgrade your project:

- 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).



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 165 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 141.

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 141.

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:

Field	Value
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:

Field	Value
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 170.

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 Extending Get-Resources scripts on page 163). 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.

```
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;
}
```

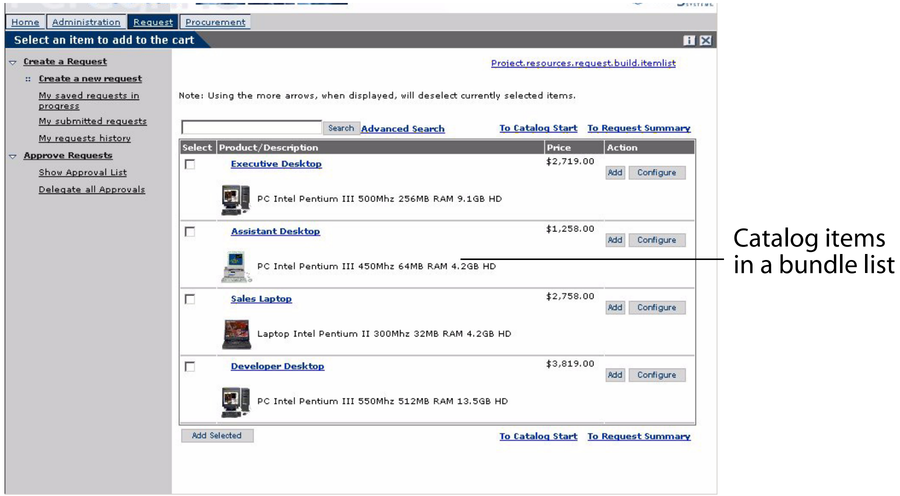
If you do not use the out-of-box subtypes in Get-Resources, you can modify your custom script by adding new subtypes to your `getLineItemSubType` function. You can use the following existing scripts as templates for subtypes.

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.



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.newcatalog.doctable	**Show Approval List** activity.

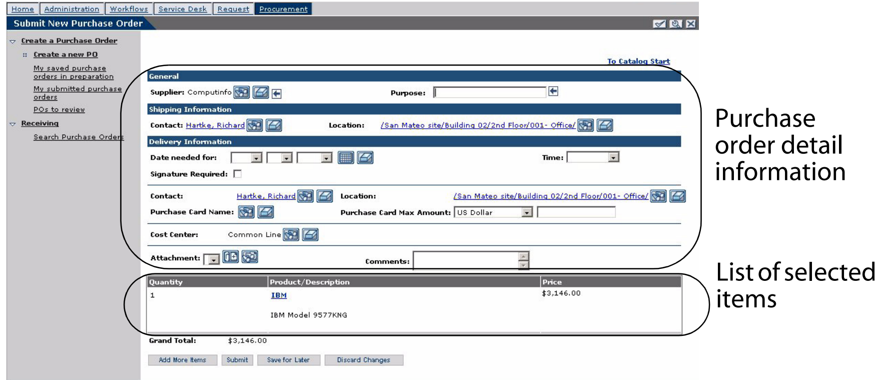
Schema used: Product

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).



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 148.

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 148.

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.

Field	Value
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:

Field	Value
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 188 and Purchase order line default values on page 194.

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).
- Looking at the details of an item from the selected item list in the **Purchase order summary** screen.
- 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

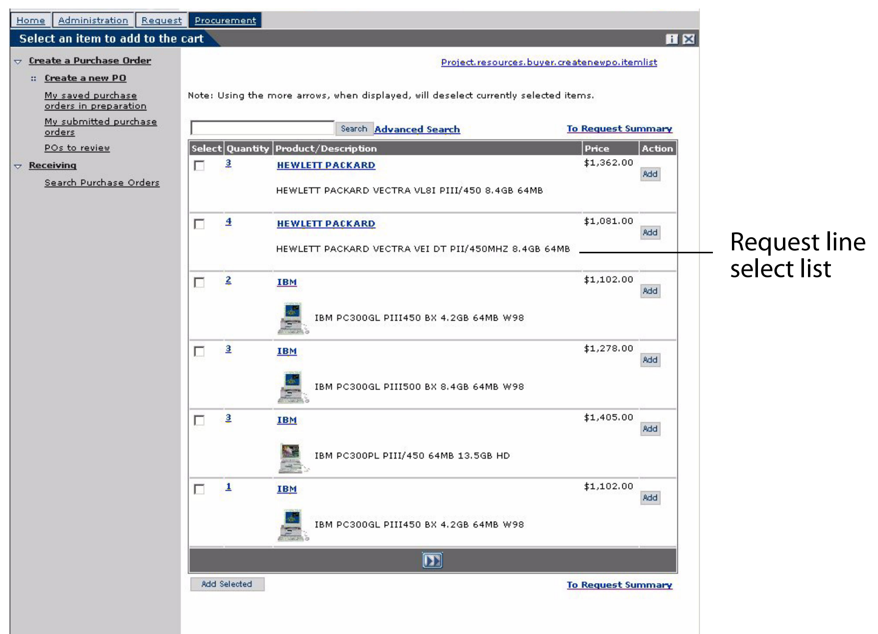
To ensure an easier upgrade, you should create a custom request interface script to add or remove subtypes (see Extending Get-Resources scripts on page 163). Within your custom script, you can use any line item field listed in the `OrderLine` schema to determine the item's subtype.

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.

```
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.



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 the Document Schema Definitions chapter.

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 add a DocExplorer Reference, you can make changes to this activity directly from the Get-Resources Web interface.

To personalize DocExplorer pages:

1 Log on to Get-Resources.

2 Click the activity name for your Document Explorer from the navigation sidebar. By default, the Document Explorer name is DocExplorer.

Important: The first time you access a Document Explorer, the interface displays a blank search form.

3 Click the wrench icon from the upper right of the interface.

4 Make your changes to the search form, and then click **Save**.

Your personalized search form displays.

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

To personalize your custom forms, add lookup fields to them. Lookup fields use some of the Personalization features found in a DocExplorer template activity.

Note: Lookup fields are already part of the DocExplorer template activity, so you need not add them there. Add lookup fields to the custom forms that you have manually built in Peregrine Studio.

You can add two types of lookup fields to your custom forms:

Lookup type	Reference
Field	See Field lookup on page 156.
Subdocument	See Subdocument lookup on page 158.

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 those values in a list. For example, when opening an asset request, 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 a lookup field.

2 Go to **New** > **Field** > **Lookup**.

3 Enter the following settings for the Data attributes.

Attribute	Description
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 is posted to the onload script when a particular lookup entry is selected.
Unique Key Field	Required if the lookup is added in a document table. Uniquely identifies the field lookup for each row of the table.

4 Enter settings for the following DocExplorer Adapter attributes.

Attribute	Description
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.

Attribute	Description
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 4.

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 its 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 are 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.

Attribute	Description
Display Field	The label you want displayed for the lookup field in the Get-Resources form. Required. If you do not enter a valid value for this parameter, an error occurs.
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 look up all other document fields in the subdocument. This value is posted to the onload script when a particular lookup entry is selected.
Unique Key Field	Required if the lookup is added to a document table. Uniquely identifies the subdocument lookup for each row of the table.

4 Enter settings for the following for the DocExplorer Adapter attributes.

Attribute	Description
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 for the path is: `schema name.subdocument name` with a period (.) between them. For example, the entry `employee.location` looks up the location subdocument from the employee schema.

5 Enter the following setting for the Link Parameters attribute.

Attribute	Description
Target form	Enter `docExplorer.documentlookup.init` 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 on page 158.

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

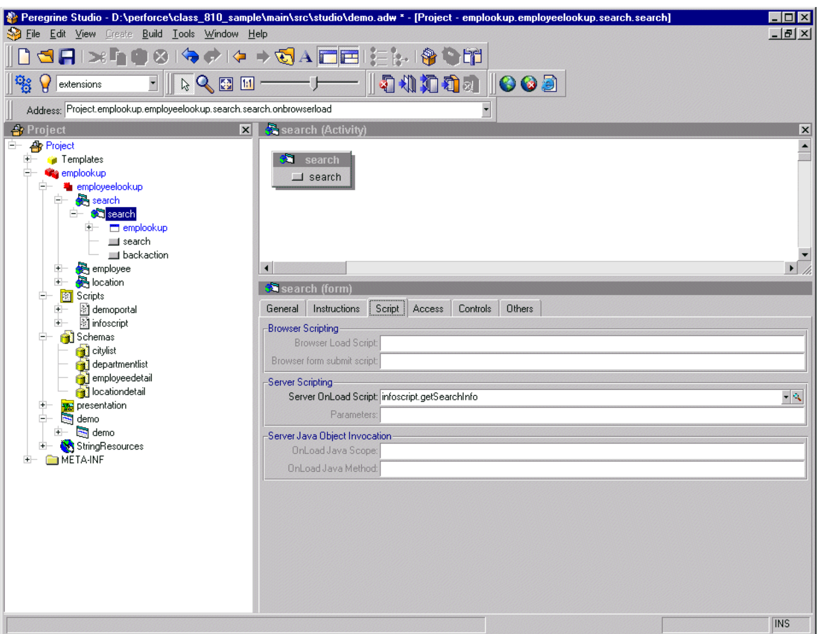
You can edit the ECMAScript in your project directly from the Peregrine Studio interface.

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.

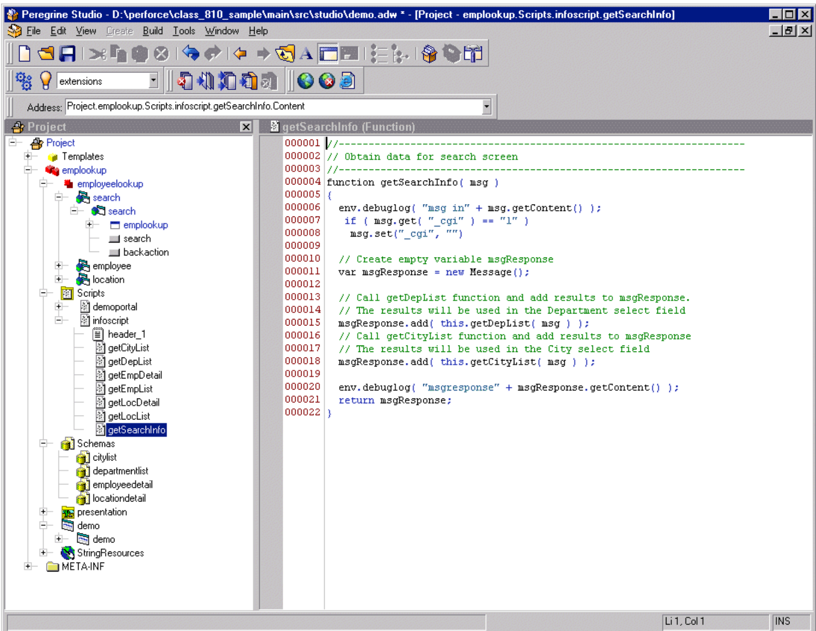
To edit an existing script:

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 script types.

Script type	Description
Form onload	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.

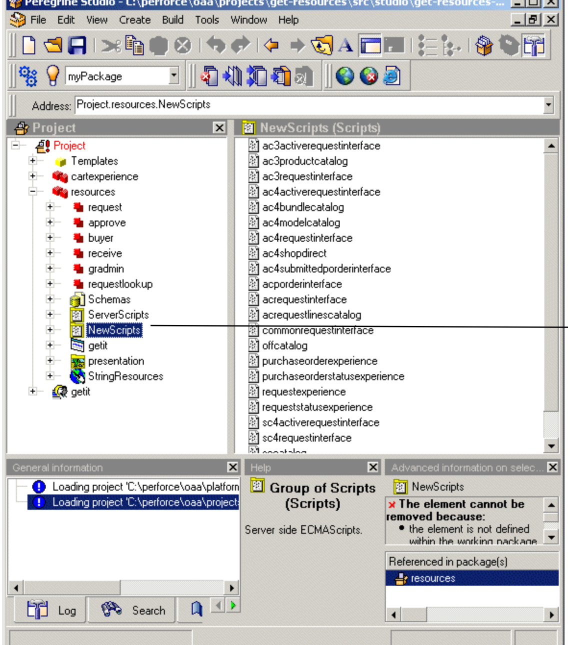
To change the request behavior:

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:

```
import sc4requestinterface;
this.__proto__ = sc4requestinterface.valueOf();
```

7 Go to the Admin Settings page, on the Get-Resources tab, and give the new script name in the **Request Interface Script** option.

8 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 142.

You can copy the existing request interface functions into your custom script.

9 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
<documents name="base">
   <document name="Product" label="Product">
      <attribute name="SupplierPartNumber" type="string"
       label="Supplier Part"/>
   </document>
</documents>

Schema extension physical mapping
<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 stores and saves 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
<documents name="base">
   <document name="RequestLine" label="Req Line" ... >
      <attribute name="SupplierPartNumber" type="string"
       label="Supplier Part"/>
   </document>
</documents>

Schema extension physical mapping
<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 rewrite with actual script extensions. For more information of creating script extensions, see Extending Get-Resources scripts on page 163.

Script setting	Value
Script to extend	catalogbase
Sample script extension	mycatalogbase

- The `mycatalog` header must import the `sccatalog` script:

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

- Add a `getNewRequestLine` function to `mycatalogbase`.

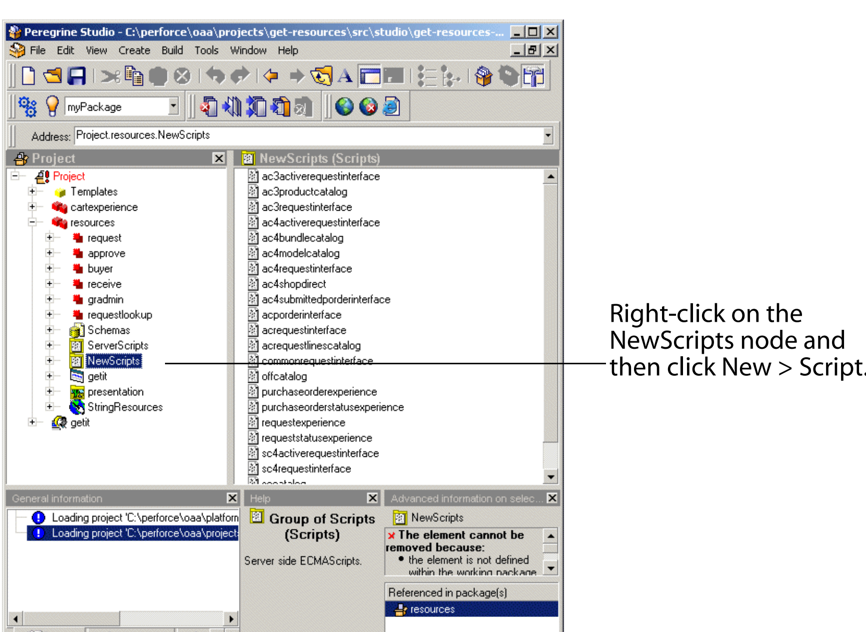
```
function getNewRequestLine(msg)
{
   // Call the parent function to build the out-of-box request
   // line document
   var msgReqLine = catalogbase.getNewRequestLine.apply
   (this, arguments);
   // Read the value of a field from the Product schema and add
   // it to a field in the RequestLine schema
   msgReqLine.add( "SupplierPartNumber", msgItem.get
   ("SupplierPartNumber"));

   return msgReqLine;
}
```

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**.



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:

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

8 Go to the Admin Settings page, on the Get-Resources tab, and give the new script name in the **Purchase Order Interface Script** option.

9 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 149.

You can copy the existing request interface functions into your custom script from one of the following scripts:

- `acporderinterface`
- `requestinterfacebase`

10 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 171.

Step 2 Create a new script that extends the identified catalog script. See Creating an extension of the existing catalog script on page 171.

Step 3 Make Get-Resources call your new script. See Calling your extended script on page 172.

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.

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 Extending Get-Resources scripts on page 163.

Script setting	Value
Script to extend	sccatalog
Sample script extension	mycatalog

- The `mycatalog` header must import the `sccatalog` script.

```
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 the default
  // requested quantity is 2 instead of 1
  msgReqLine.set("Quantity", "2", false);

  return msgReqLine;
}
```

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 Extending Get-Resources scripts on page 163.

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;
}

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;
}
```

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:

Layer	Description
ActivityCartExperience template	Defines the screen flow.
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.
requestinterfacebase script (and any scripts you create to extend it)	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.
catalogbase script (and any scripts you create to extend it)	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.

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 the following.

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: `grhelper.getUniqueId` 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 the Peregrine Studio Projects and Packages chapter.

Step 2 Add a new schema file to your Peregrine Studio project. See Adding a schema to your Peregrine Studio project on page 179.

Step 3 Add logical and physical mappings to your schema file. See Adding logical and physical mappings to your schema on page 180.

Step 4 Configure other project components to use your custom schema. See Tailoring forms and components on page 119.

Step 5 Rebuild your Get-Resources project. See the Peregrine Studio Projects and Packages chapter.

Step 6 Deploy your new Get-Resources project files. See the Peregrine Studio Projects and Packages chapter.

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 must meet the following criteria:

- The schema name is in all lower case.
- 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 181.

Step 2 Add two `<documents>` elements for the logical mappings. See Adding logical mapping <documents> elements on page 181.

Step 3 Add two `<document>` elements to define the schema name. See Adding logical mapping <document> elements on page 181.

Step 4 Add one <attribute> element for each logical mapping you want to create. See Adding logical mapping <attribute> elements on page 182.

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 the Document Schema Definitions chapter.

4 Add any optional attributes to the `<attribute>` elements.

For more information about the attributes available for the `<attribute>` element, see the Document Schema Definitions chapter.

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 184.

Step 2 Add two `<document>` elements to define the back-end database table name. See Adding physical mapping <document> elements on page 185.

Step 3 Add one `<attribute>` element for each logical mapping you created. See Adding physical mapping <attribute> elements on page 185.

Adding physical mapping <documents> elements

1 Add another set of `<document>` elements between the `<schema>` element containers:

```
<?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>
```

These elements are the container for the physical mappings.

2 Add the `name` attribute to the `<documents>` element:

```
<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:

```
<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 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>
```

These elements are the container for the back-end database table to be queried.

2 Add the `name` attribute to the `<document>` element:

```
<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 the Document Schema Definitions chapter.

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 Schema Definitions in the Get-Resources Administration Guide.

Sample schema

The following is a sample schema that you can use for as a template for your own custom schemas.

```
<?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="acsc">
      <document name="sample" table="amRequestincidentsamContact"/>
         <attribute name="Id" field="lReqIdincident.idlContactId" />
         <attribute name="contact" field= "lEmplDeptIdcontact.nameName"/>
      <document/>
   </documents>

</schema>
```

Adding data validation

You can have Get-Resources 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.

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:

Option	Action
Subdocument	Select the **Required** option under the Explorer Options section.
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 Systems recommends that you extend the `validateRequest` function with a custom script rather than updating the function directly. For

more information of extending scripts, see Extending Get-Resources scripts on page 163.

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 Extending Get-Resources scripts on page 163 for more information about this sample custom script.

```
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;
}
```

The `validateRequestEx` function performs extra validation to prevent submitting an empty request or a purchase order. It ensures that a request or a purchase order has at least one line item before it is submitted. The `saveRequest` function calls this function before saving a request. The parameters are the entire message and 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 has no line item.

Peregrine recommends that you extend the `validateRequestEx` function with a custom script if you want to add more validation checking rather than updating the function directly.

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 188 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 Systems recommends that you extend the `getRequestDefaultValues` function with a custom script rather than updating the function directly. For more information of extending scripts, see Extending Get-Resources scripts on page 163.

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 Extending Get-Resources scripts on page 163 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;
}
```

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 Systems recommends that you extend the `getRequestDefaultValues` function with a custom script rather than updating the function directly. For more information of extending scripts, see Extending Get-Resources scripts on page 163.

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.

```
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 values.
               msgReqLine.remove("EndUser", false);
               if (msgEndUser)
                  msgReqLine.add(msgEndUser);
            }
         }
      }
   }
   return msgRequest;
}
```

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 190 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

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, see Setting request line default values from catalog entries on page 170.

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, see Setting request line default values to values in a request on page 192.

Translating tailored modules

Out-of-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 195.

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 198.

To configure Get-Resources to use your new translation, refer to Configure Get-Resources to use new string files on page 199.

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 has the ISO-639 two letter abbreviation EN in the file name.

All strings files have the STR file extension.

2 Search for any new text that you added to your tailored Peregrine Studio project.

The string file uses the following format:

```
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 are 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 has the ISO-639 two letter abbreviation EN in the file name.

   All strings files have the 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 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 is the default language used. The other languages you define are available in a drop-down list.

5 In the **Content type encoding** field, enter the character encoding 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 Peregrine Portal** to implement your changes.

Users can select the display language for their session used when they login to the Peregrine OAA Platform.

A CHAPTER Troubleshooting and FAQs

This appendix contains troubleshooting information for Peregrine Studio and tailoring tasks.

This chapter covers the following topics:

- Get-Resources environment on page 202
- Peregrine Studio on page 203
- Scripting errors on page 206
- Tailoring errors on page 207

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 – install JRE

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.

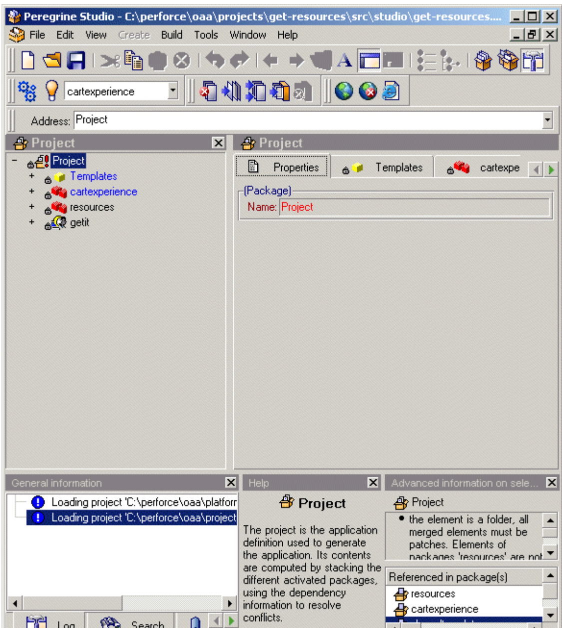
Peregrine Studio

This section describes common problems with write protections, conflicts, and build errors generated with Peregrine Studio.

Cannot edit — components have 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 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 displays 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.

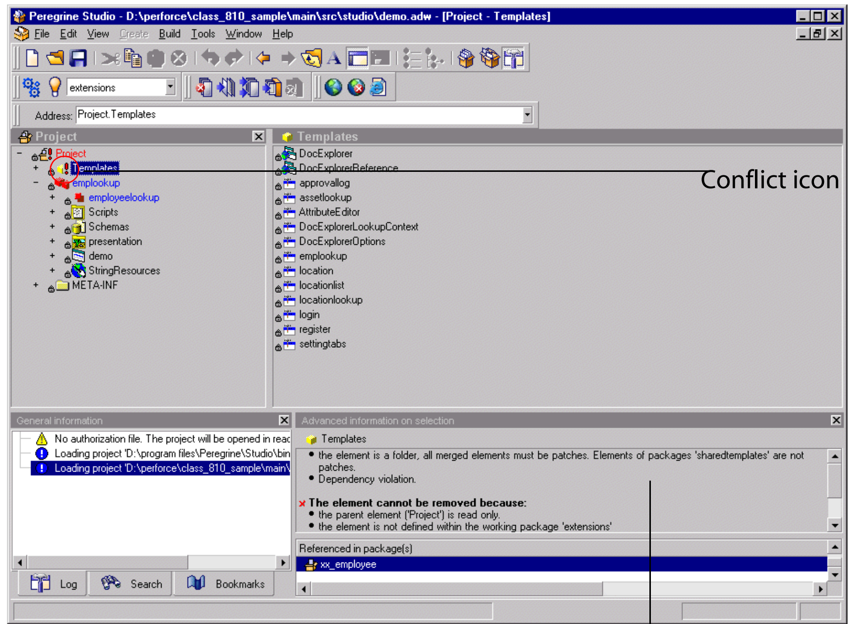
Solution

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**. Peregrine Studio displays a new information window at the bottom of the interface. This window displays information on the conflict.

The information on selection indicates whether you have a resource or a dependency conflict.



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 displays when you select a form:

```
Unable to find script file for <name>
```

This message also appears in the archway.log file.

Solution

This error message is usually the result of an invalid script file name or adapter name.

Verify in Peregrine Studio that the form is calling a valid script file name. In particular make sure that the script name does not use mixed case. Script file names should be in all lower case. If you copied a script from another form or Web application you may have renamed the script incorrectly.

Verify that the script calls a valid adapter. 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.

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 displays when you select a form:

```
ECMAScript Error: Error Message: Runtime error Function called on
undefined value or property
```

This error also displays 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 displays when you select a form:

```
ERROR:...: ***SQL Exception caught***
```

The script output displays the following error:

```
-3010: [...][...] Too few parameters. Expected 1.
```

These messages 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 schemas 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 also displays in the `archway.log` file.

Solution

The schema name you defined for the form is wrong. The schema name can 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.

APPENDIX B

Copyright Notices

Peregrine Systems acknowledges the copyrights belonging to the following third parties. (This appendix constitutes a continuation of the copyright page.)

Notices

finj.jar

The `finj.jar` is covered under the GNU Lesser General Public License version 2.1. The license is included below under section GNU LGPL.

The finj source is included in `finj.jar` file in the `\Program Files\Peregrine\Studio\bin\classes` directory.

GNU LGPL

GNU LESSER GENERAL PUBLIC LICENSE

Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc. 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to

guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to somespecially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to freesoftware only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived

from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License").

Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to theentire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or

distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License.

Section 6 states terms for distribution of such executables. When a "work that uses the Library" uses material from a header filethat is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work

can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law. If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object

file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompany is the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.

b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Librarysubject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted incertain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR

CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

<one line to give the library's name and a brief idea of what it does.>

Copyright (C) <year> <name of author>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the library `Frob' (a library for tweaking knobs) written by James Random Hacker.

<signature of Ty Coon>, 1 April 1990 Ty Coon, President of Vice

That's all there is to it!

Index

PEREGRINE

A

actions. *See* form components
activity component 41
adding
    subdocument lookups 158
Archway
    scripts 85
AssetCenter 153
authorization file
    Peregrine Studio 23

B

bookmarks, adding in Studio 30
build options 46–47
    build directory 47
    character encoding 47
    EJB user 47
    exclude files 47
    presentation folder 47
    temporary directory 47

C

cart experience 173
cascading style sheets 40
component template 67, 68
components
    group of files component 42
    group of modules component 41
    group of schemas component 42
    group of scripts component 42
    group of strings component 42
    hierarchy of 40
    in Peregrine Studio 55
    module component 41
    relationships among 41–42
conflicts
    defined 52
    resolving 52–53, 205
creating
    package extensions 49
    schemas 179
customer support 14

D

data validation
    for purchase order 190
    for request summary 188
    methods of 187
    tailoring tasks 118
    validateRequestEx 189
dates, manipulating in scripts 91
default values
    for purchase order 194
    for purchase order line 194
    for request line 170
    for request summary 190
    tailoring tasks 119
dependencies
    setting for packages 51
dependency conflicts. *See* Conflicts
deployment directory 47
development environment

requirements for 25
DocExplorer Reference
adding 154
DocExplorers
tailoring tasks 118
Document field
format of names 128

E

ECMAScript 84
errors
sysntax error in FROM clause 209
too few parameters 208
Unable to find script file 206
undefined value or property 207

F

field labels, changing 123
field lookup 156
fields
making required 187
fields. *See* form components
fieldsection component 68
form component 42
form components
action 42, 81
changing schemas 126
common 67
component template 67
date picker 58
described 42
document table 77, 78
field form components 123
fields 42
fieldsection 68, 70
form columns 80
hidden data field 73, 74
hiding 124
labels 123
lookups 42
making read-only 125
names in 128–130
redirection 74
selectbox 71
simple table 75, 76
table link 78
tables 42
tailoring 119–120
tailoring tasks 117
text columns 79
text edit 70, 71
forms
changing instructions 121
changing onload scripts 122
changing titles 120
server-side 85–86
tailoring tasks 117
framesets
displaying forms in 130

G

Get-Resources forms 138–152
catalog select list 144
purchase order line detail 149
purchase order summary 146
request line detail 142
request line selection 151
request summary 139
group of scripts component 42

H

HTTP Listener 34
HTTP listener
enabling in Peregrine Studio 34

I

installation
tailoring kit 17
instructions, changing in forms 121
interface components. *See* Form components 42
ISO character encoding. *See* character encoding

J

JavaDocs 114
JavaScript 84

L

lookup fields
adding 156
subdocument lookups 158

lookups. *See* form components

M
messages, scripts 97

N
nodes 31, 204
- group of schemas node 179

O
onload scripts
- changing in forms 122
- defined 122

P
package extensions 48–51
packages
- activating 50
- deactivating 50
- defined 48
- dependencies 50, 51

Peregrine Studio
- authorization file 23

Peregrine Systems customer support 14
Personalization
- lookup fields 156
- requirements 153
- with DocExplorers 152

Portal components
- creating 134

presentation files 40
Project Explorer 31
projects
- *See also* Web applications
- components of 39
- conflicts within 53
- files within 43

R
resource conflicts. *See* conflicts
Rhino JavaScript Debugger 92–94

S
schema template example 187
schemas
- adding logical and physical mappings 180
- changing in form components 126
- creating 179
- creating your own 178
- document fields 127
- sample 187
- tailoring tasks 118
- testing from a URL 95–96
- using with DocExplorers 153

scripts
- adding to Peregrine Studio project 89, 162
- cartexperience 175
- catalog 177
- client-side 84
- creating XML message objects 97
- displaying variables in form components 133
- ECMAScript 84
- editing 87, 159
- extending the request interface script 163–170
- extensions of 163
- format of variables 133
- JavaScript 84
- list of references 113
- object oriented usage 100
- onload scripts 85–86
- prototype property 100
- request interface 176
- roles of 85
- samples 105–111
- server scripts 84
- server-side 84
- tailoring tasks 118
- testing from a URL 94
- uses for 83

ServiceCenter 153
source files
- opening in Peregrine Studio 23

string files
- translating 196, 198

subdocument lookup field 158

T
tables. *See* form components
tailoring

common form components 67
form components 119–120
tailoring kit
installation 17
tailoring tasks 117
technical support 14
templates
ActivityCartExperience 174
templates component 41
testing environment
requirements for 25
titles, changing in forms 120
translating
tailored modules 195
troubleshooting
cannot start Java 202
conflicts 204
JRE must be installed 202
Read-only components 203
redirections 209
script error 206
script error Unable to find script file 206
script error undefined value or property 207
sysntax error in FROM clause 209
too few parameters 208
virtual memory error 202

U

UNIX
deploying tailoring changes to 54
URL
querying scripts and schemas from 94

V

validateRequestEx 189
variables
referring to XML attributes 134
visible flag
hiding form components 124

W

Web applications
viewing changes 36

X

XML
creating message objects from scripts 97
example of Document field names 129
example of script variable name 133
viewing source code 33

1/20/05