

Peregrine

Get-Services

Tailoring Kit Guide

Version 4.0.1—For Windows

Copyright © 2002 Peregrine Systems, Inc. or its subsidiaries. All rights reserved.

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®, and ServiceCenter® are registered trademarks of Peregrine Systems, Inc. or its subsidiaries.

This product includes software developed by the Apache Software Foundation (<http://www.apache.org/>) and by Advantys (<http://www.advantys.com>). This product also contains software developed by Sun Microsystems, Inc. and Netscape Communications Corporation.

This document and the related software described in this manual are supplied under license or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. 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 have comments or suggestions about this documentation, please send e-mail to support-sd@peregrine.com

This edition applies to version 4.0.1 of the licensed program.

Peregrine Systems, Inc.
Worldwide Corporate Headquarters
3611 Valley Centre Drive San Diego, CA 92130
Tel 800.638.5231 or 858.481.5000
Fax 858.481.1751
www.peregrine.com



Contents

	Introducing the Get-Services Tailoring Kit	9
	About this Guide	11
	Conventions Used in this Guide.	11
Chapter 1	Installing the Get-Services Tailoring Kit	13
	Installation Requirements	14
	Installing the Get-Services Tailoring Kit	14
	Opening the Get-Services Project	18
	Setting up a tailoring environment.	18
	Setting up a Development Environment	19
	Setting Up a Testing Environment.	19
Chapter 2	Tailoring Tasks	21
	Tailoring Tasks independent of Peregrine Studio.	22
	Personalization.	22
	Schema extensions	22
	Tailoring tasks requiring Peregrine Studio	23
	Forms and Form components	23
	DocExplorers	23
	Scripting	23
	Schemas.	24
	Data validation	24
	Default values	24

Chapter 3	Using Peregrine Studio	25
	The Peregrine Studio interface	26
	Project Explorer	27
	Drag and drop	29
	Enabling the HTTP Listener and Form Information	30
	Viewing XML source code	32
	Finding changes indicated with color text	32
Chapter 4	Peregrine Studio Projects and Packages	35
	Peregrine Studio projects	36
	Project components	37
	Project component descriptions	37
	Project files	40
	Building a project	41
	XML to JSPs	41
	Build options	42
	Setting project build settings	42
	Peregrine Studio project packages	44
	Saving changes with package extensions	45
	Activating and deactivating packages	45
	Package dependencies	46
	Setting package dependencies	46
	Warnings for conflicts	47
	Deploying tailoring changes	49
	Deploying to Windows platforms	49
	Deploying to UNIX platforms	49
	Translating tailored modules	49
	Editing existing translation strings files	51
	Adding new translation strings files	52
	Configure Get-Services to use new string files	53
	Adding Get-Services to an existing frameset	54
Chapter 5	Forms and Form Components	55
	Tailoring forms	56
	Changing a form's title	57
	Changing a form's instructions	58

Changing a form's onload script	59
Changing a form component's label	60
Hiding a form component	61
Changing a form component to read-only	62
Changing the schema that a form component uses	63
Changing the document field that a form component uses	64
Displaying a form within a frameset	67
Displaying a script variable in a form component	69
Creating a portal component	72
Types of form components	74
Component template containers	74
Fieldsection containers	75
Text edit fields	76
Selectbox fields	77
Hidden data fields	79
Redirections	80
Simple table	81
Table links	82
Text columns	82
Actions	83
Chapter 6 Adding Personalization Functionality	85
Supporting personalization	86
Activating personalization	87
Making a schema visible to portal components	88
Personalizing with DocExplorers	89
DocExplorer forms and functions	89
Adding a DocExplorer reference	90
Personalizing a DocExplorer reference	91
Adding Personalization to lookup fields	92
Using the personalization interface.	95
Adding an existing field to a personalized form	97
Removing a field from a personalized form	97
Personalizing a field attribute	98

Chapter 7	Scripting	99
	How scripts are used.	100
	Types of Scripts.	100
	Where Scripts are Stored.	101
	How Scripts are Used	102
	Editing an existing script	105
	Adding a custom script	107
	Testing Scripts	109
	Rhino JavaScript Debugger	109
	URL queries	110
	Common Message Operations	113
	Using ECMAScript in an Object Oriented manner	116
	ECMAScript implementation in Get-Services	116
	Name resolution in ECMAScript	116
	Using the object prototype for object oriented programming	116
	How to use object orientation for tailoring	120
	Sample Scripts	121
	General Script Samples	121
	Selecting a Field from a Schema	121
	Calling Other Scripts and Combining the Results	123
	Form Script Sample	125
	Creating an XML Document from a Schema	125
	References	128
	Sources for Client-side JavaScript	128
	JavaDocs for the Main Archway Package	128
Chapter 8	Document Schema Definitions	129
	Understanding Document Schema Definitions	130
	How to use schemas	131
	Schema extensions	132
	When to use schema extensions	132
	Creating schema extensions	133
	Identifying the schema to extend	133
	Locating the schema on the server	134
	Creating the schema extension target folders and files	135

Editing the schema extension files	136
Adding a new field to the Available Fields list	136
Hiding an existing field from the Available Fields list	138
Changing the label a field displays in the Available Fields list.	139
Changing the list of forms where a field is visible.	140
Changing the physical mapping of a field	142
Changing the type of form component a field uses	143
Adding subdocuments to the Available Fields list	144
Creating custom schemas.	148
Adding a schema to your Peregrine Studio project	149
Adding logical and physical mappings to your schema	149
Sample schema	155
Schema Elements And Attributes	156
<?xml>	156
<schema>	156
<documents>	156
<document>.	158
<attribute>	162
<collection>	166
Documents	168
Subdocuments	169
Chapter 9 Using Get-Services Tailoring	175
Adding data validation to fields	176
Making a field required	176
Adding data validation with a custom script function.	177
Assigning default values to fields with a custom script function.	185
Changing the strings displayed by priority, severity, or status fields	192
Removing display values for priority, severity, or status	195
Appendix A Peregrine Studio Components	199
Appendix B Troubleshooting and FAQs	211
Get-Resources Environment	212
Out of memory error	212
Cannot start Java – JRE must be installed	212

Peregrine Studio	213
Cannot edit — components are displayed with grey background	213
Red exclamation point (conflict icon) displayed next to nodes	214
Scripting Errors.	216
Unable to find script file	216
Script produces an ECMAScript error	217
ECMAScript error: undefined value or property	217
Tailoring Errors.	218
Wrong start form is displayed for activity.	218
Script output not appearing in form component.	218
Too few parameters error	219
Get-Services always goes to redirection form	220
Syntax error in FROM clause	220
Index	221

Introducing the Get-Services Tailoring Kit

The Get-Services Tailoring Kit includes:

- Peregrine Studio
- Source files for Get-Services

The OAA Tailoring Kit 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-Services. Get-Services consists of a series of Web-based interfaces that allow users to, for example, open tickets, assign tickets to IT employees, and view historical information on incidents. The Peregrine Portal common interface determines what portions of Get-Services the user sees.

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

- A collection of Java Server Pages (JSPs) that provide the browser interfaces for Get-Services. The Get-Services JSP content is created during the Studio build process.
- A Web server to host the Get-Services JSP content.
- A Java-enabled application server to run the Archway servlet. The Archway servlet routes and formats data requests between Get-Services and the back-end database.
- A collection of ECMAScripts that allow for dynamic parsing and formatting of Get-Services data sent to and received from the client Web browser.

From an administrative perspective, Get-Services are the output of one or more Peregrine Studio project files. Studio elements such as packages, modules, activities, and forms describe the end-user interface. Other Peregrine Studio elements such as ECMAScripts and document schema definitions determine what data the Get-Services interface receives or processes from back-end database.

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

- A project file that describes Get-Services. Each project file contains only the code necessary to produce and deploy Get-Services.
- Components that define the functionality of Get-Services. The Get-Services project is built from packages, modules, activities, forms, and form components. Each of these project components is saved as an XML file in the Peregrine Studio project.
- A back-end database or application to store the data accessed by Get-Services 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-Services forms.

About this Guide

This guide is intended for use by a developer who will be tailoring Get-Services.

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

- The Get-Services installation and administration 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-Services forms are shown using the Classic theme.

The following documentation conventions are used in this guide:

Object	Example
Button	Click Next
File name	The <code>login.jsp</code> file
Sample script or XML code	<pre>var msgTicket = new Message("Problem");</pre> <p>...</p> <pre>msgTicket.set("_event", "epmc");</pre> <p>The ellipsis (...) is used to indicate that portions of a script have been omitted because they are not needed for the current topic. Samples of code are not entire files, but they are representative of the information being discussed in a particular section.</p>
Menu option	Select Start>Program Files.
Book title	Refer to the <i>Get-Services Installation Guide</i> .

1 Installing the Get-Services Tailoring Kit

CHAPTER

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

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

This chapter covers the following topics:

- *Installation Requirements* on page 14
- *Installing the Get-Services Tailoring Kit* on page 14
- *Opening the Get-Services Project* on page 18
- *Setting up a tailoring environment* on page 18

Installation Requirements

You can install the Get-Services Tailoring Kit on any system that meets the following requirements:

- Windows operating system
- Java 2 SDK Standard Edition version 1.3.1_05
- Peregrine Studio installed

You can install the Java 2 SDK and Peregrine Studio from the Get-Services Tailoring Kit installation CD.

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

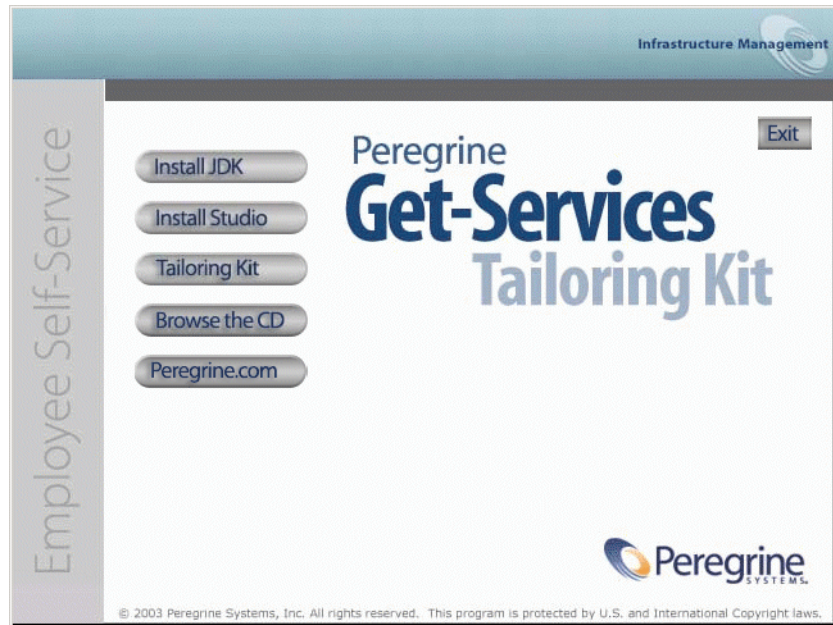
Installing the Get-Services Tailoring Kit

The following sections describe how to install the Get-Services Tailoring Kit.

To install the Get-Services Tailoring Kit:

- 1 Insert the Get-Services Tailoring Kit installation CD into the CD-ROM drive.

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

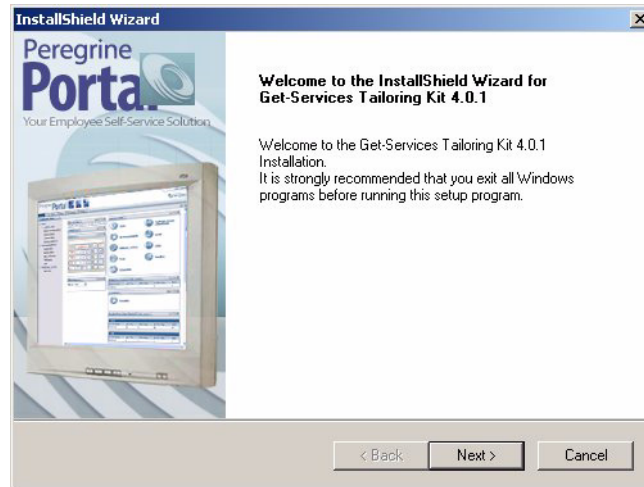


- 2 Install any required components for the Get-Services Tailoring Kit.
 - **Install JDK.** Click this button to install the Java 2 SDK on your system.

Warning: You must install a JDK if you have installed Get-Services with Tomcat.

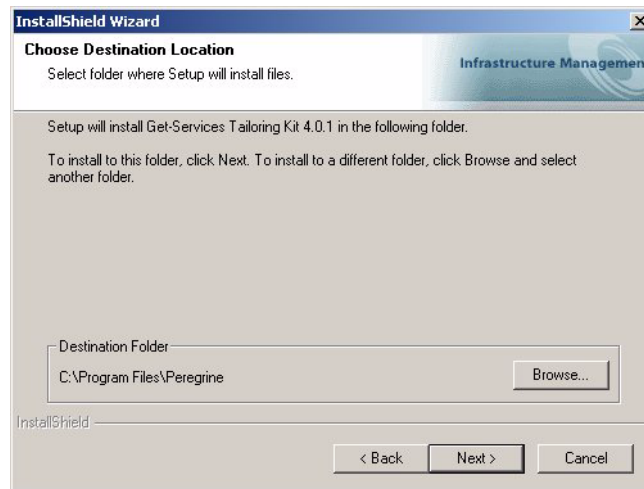
- **Install Studio.** Click this button to install Peregrine Studio on your system.
- 3 Click Tailoring Kit.

The Get-Services Tailoring Kit installer opens.



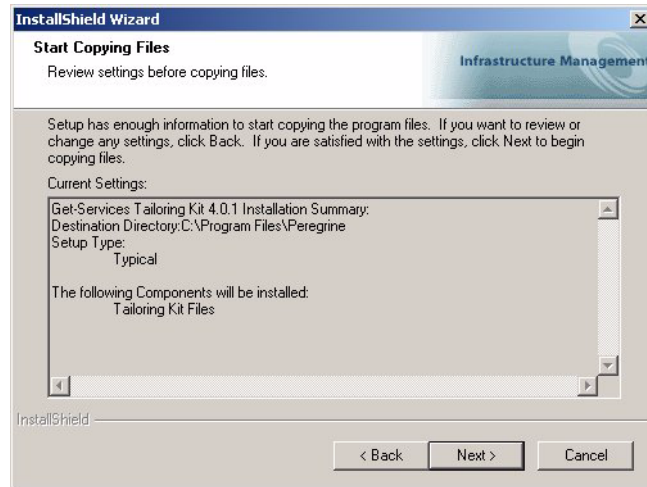
- 4 Click Next to continue.

The Choose Destination Location page opens.



- 5 Click Next to accept the default installation location, or click **Browse** to select another installation location, and then click Next to continue.

The Start Copying Files page opens.



- 6 Verify that the information is correct, and then click **Next**.

The installer copies and deploys the files to your system and then the InstallShield Wizard Complete page opens.



- 7 Click **Finish** to close the InstallShield Wizard.

Opening the Get-Services Project

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

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

To open the Get-Services project in Peregrine Studio:

- 1 Click **Start > Programs > Peregrine > Studio > Peregrine Studio**.
Peregrine Studio opens.
- 2 Click **Tools > Authorization file**.
- 3 In any text editor, open the authorization file provided for Peregrine Studio.
- 4 Copy the contents of the authorization file into the Authorization file dialog box in Peregrine Studio. Click **OK**.
- 5 Click **File > Open project**.
- 6 Browse to the location of your Get-Services project file (.adw file). For example:
`C:\Program Files\Peregrine\Get-It Tailoring Kit\get-services`
- 7 Click **Open**.
- 8 Create a new package to save your changes. See *Saving changes with package extensions* on page 45.

Setting up a tailoring environment

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

Setting up a Development Environment

You need the following minimum components for a Get-Services 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-Services source files.
- Java Development Kit 1.2.2 or later if you want to create or edit your own wizards for Peregrine Studio.

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

- Build your Get-Services projects on the development computer and copy the results to a deployment computer

or

- Enter the network path to the deployment computer 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 Project Settings to check out and check in the source files.

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-Services source files.
- If you want to create or edit your own wizards for Peregrine Studio, you will need to install a Java Development Kit 1.2.2 or later. The Java 2 SDK Standard Edition v1.3.1_05 is provided on the installation CD for your Web application.
- An installed instance of Get-Services.

- Web server (necessary to serve Get-Services JSP content).
- Java-enabled application server (necessary to run the Archway servlet). Tomcat is provided on the Get-Services installation CD.
- JavaScript-enabled Web browser (necessary to view changes to Get-Services).

With this testing environment, you can build and view your changes from a single computer. To set up a testing environment, perform a full installation. Refer to the *Get-Services Installation Guide* for instructions and requirements.

Peregrine Studio also allows you to save multiple versions of our changes in separate project files. When you are ready to test a particular tailoring change, you can load the project, build it, and deploy it to your test environment.

2 Tailoring Tasks

CHAPTER

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

This chapter covers the following topics:

- *Tailoring Tasks independent of Peregrine Studio* on page 22
- *Tailoring tasks requiring Peregrine Studio* on page 23

Tailoring Tasks independent of Peregrine Studio

The following tailoring task can be done outside of Peregrine Studio.

Personalization

You can perform the following tailoring tasks using the on-screen personalization interface:

- *Activating personalization* on page 87.
- *Making a schema visible to portal components* on page 88.
- *Adding an existing field to a personalized form* on page 97.
- *Removing a field from a personalized form* on page 97.
- *Personalizing a field attribute* on page 98.

Schema extensions

You can perform the following tailoring tasks by creating schema extensions:

- *Creating schema extensions* on page 133
- *Adding a new field to the Available Fields list* on page 136
- *Hiding an existing field from the Available Fields list* on page 138
- *Changing the label a field displays in the Available Fields list* on page 139
- *Changing the list of forms where a field is visible* on page 140
- *Changing the physical mapping of a field* on page 142
- *Changing the type of form component a field uses* on page 143
- *Adding subdocuments to the Available Fields list* on page 144

Tailoring tasks requiring Peregrine Studio

The following tailoring tasks require Peregrine Studio.

Forms and Form components

You can use Peregrine Studio to tailor forms and form components in the following ways:

- *Changing a form's title* on page 57.
- *Changing a form's instructions* on page 58.
- *Changing a form's onload script* on page 59.
- *Changing a form component's label* on page 60.
- *Hiding a form component* on page 61.
- *Changing a form component to read-only* on page 62.
- *Changing the schema that a form component uses* on page 63.
- *Changing the document field that a form component uses* on page 64.
- *Displaying a form within a frameset* on page 67.
- *Displaying a script variable in a form component* on page 69.
- *Creating a portal component* on page 72
- *Changing the strings displayed by priority, severity, or status fields* on page 192
- *Removing display values for priority, severity, or status* on page 195

DocExplorers

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

- *Adding a DocExplorer reference* on page 90.
- *Personalizing a DocExplorer reference* on page 91.
- *Adding Personalization to lookup fields* on page 92.

Scripting

You can use the following scripting methods for tailoring:

- *Editing an existing script* on page 105
- *Adding a custom script* on page 107

- *Common Message Operations* on page 113

Schemas

You can use Peregrine Studio to tailor schemas in the following ways:

- *Creating custom schemas* on page 148
- *Adding a schema to your Peregrine Studio project* on page 149
- *Adding logical and physical mappings to your schema* on page 149

Data validation

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

- *Adding data validation to fields* on page 176
- *Adding data validation with a custom script function* on page 177

Default values

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

- *Assigning default values to fields with a custom script function* on page 185

3 Using Peregrine Studio

CHAPTER

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

This chapter covers the following topics:

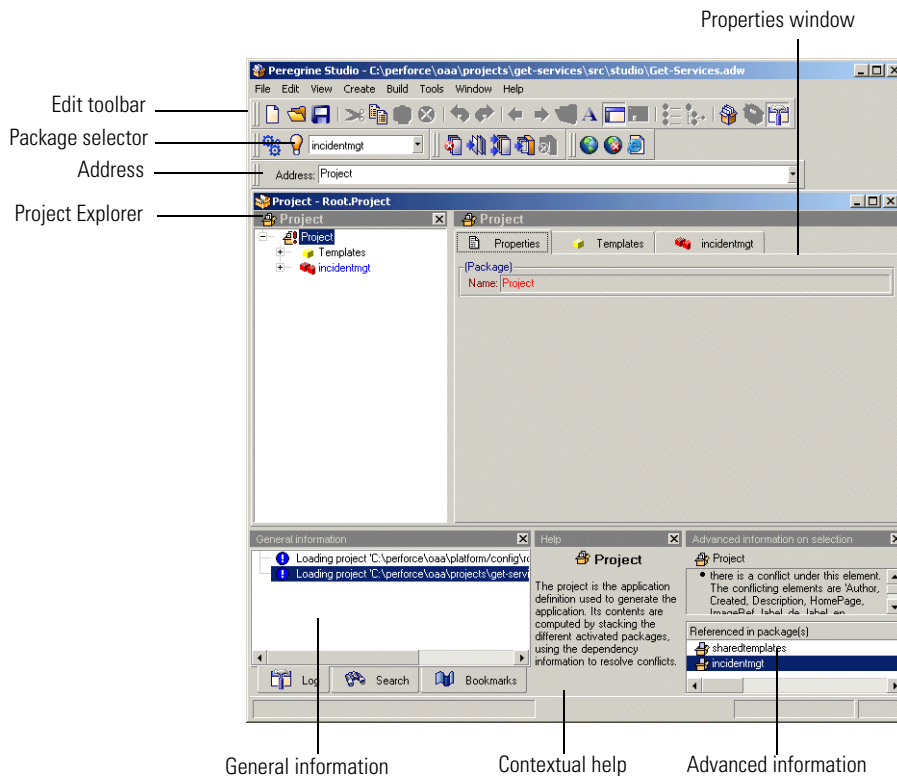
- *The Peregrine Studio interface* on page 26
- *Enabling the HTTP Listener and Form Information* on page 30
- *Viewing XML source code* on page 32
- *Finding changes indicated with color text* on page 32

The Peregrine Studio interface

The Peregrine Studio interface includes:

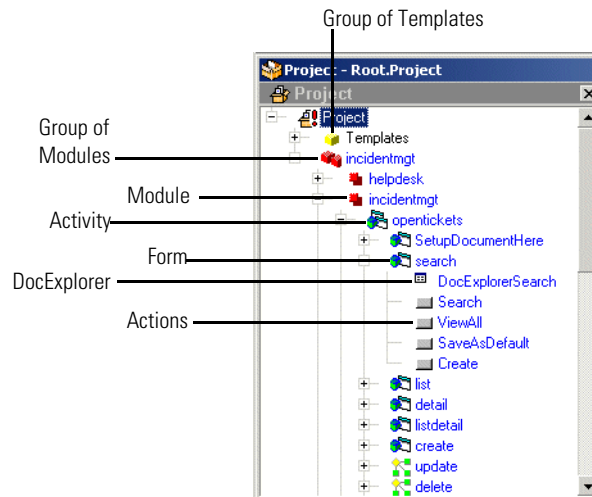
- Project Explorer
- Properties window
- Edit toolbar
- General information display
- Contextual help
- Address
- Package selector
- Advanced information

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



Project Explorer

The Project Explorer provides a hierarchical 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.

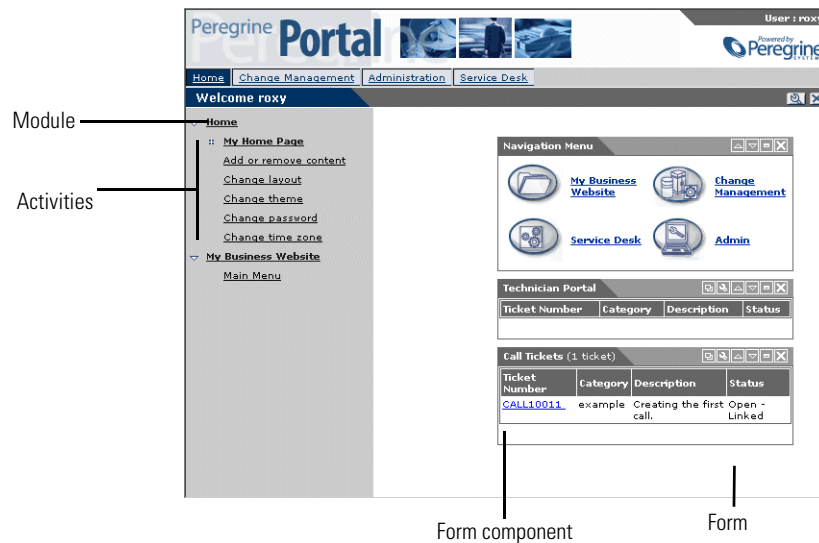
Right-click a node

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

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



The address bar

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

To display the address bar:

- 1 Open Peregrine Studio.
- 2 Click View > Address Bar.

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.

Enabling the HTTP Listener and Form Information

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-Services with a browser and quickly bring up any particular form that needs modification.

Important: The HTTP Listener cannot bring up certain forms that are built using DocExplorer as the source code is no longer provided with the Get-Services tailoring kit.

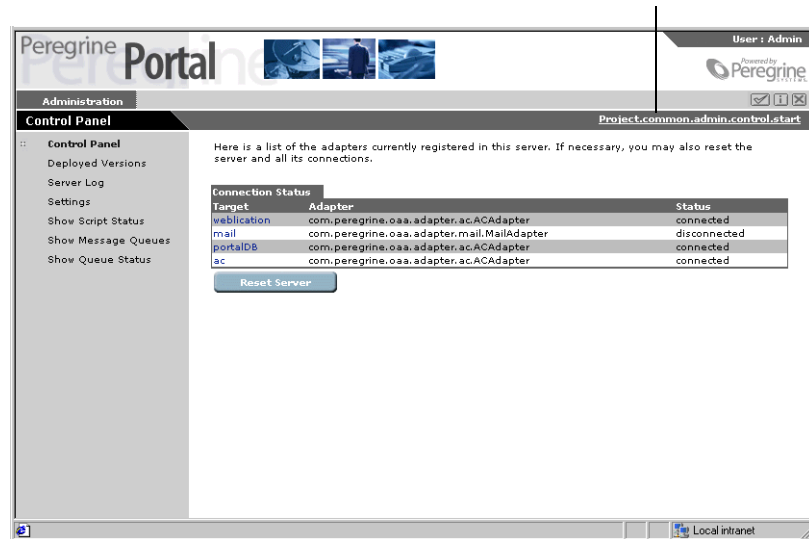
The HTTP Listener is best used to tailor forms that are built from Peregrine Studio form components.

To enable the HTTP Listener and Form Information functionality:

- 1 Enable the HTTP listener as follows:
 - a Open Peregrine Studio.
 - b Click Tools > Options.
 - c Select the **Use listener** check box from the HTTP Listener section.
 - d Select the port number you want the HTTP listener to use (the default port is 81), and then click OK.
 - e Save your Peregrine Studio project.
 - f Close and then re-open Peregrine Studio to initialize the HTTP listener.
- 2 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. See *Saving changes with package extensions* on page 45.


- 3 Log in to Get-Services as an administrator, or access the Admin module directly from the Administrator login page (`admin.jsp`).
- 4 Click Admin > Settings to display the Settings form.
- 5 On the **Logging** tab, set the **Show form info** setting to `true`.
- 6 Click **Save** at the bottom of the form to activate your new settings.
- 7 On the Control Panel form of the Admin module, click **Reset Server** to commit your changes.

Click this link to open the form in Studio.




- 8 Navigate to the form you want to tailor.
 - 9 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.

Viewing Referenced Components

Whenever an item links to or references another component, Peregrine Studio will display a magnifying glass 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.

Viewing XML source code

All project information is saved in XML files that you can see from Peregrine Studio. Peregrine Studio does not support direct editing to the XML source of components. All XML source views are listed with a grey background which indicates that the item is read-only.

To view the XML source code:

- 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.

Finding changes indicated with color text

All changes or additions you make to your Peregrine Studio project are highlighted with blue text.

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.

4 | Peregrine Studio Projects and Packages

CHAPTER

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

This chapter includes the following topics:

- *Peregrine Studio projects* on page 36
- *Building a project* on page 41
- *Peregrine Studio project packages* on page 44
- *Warnings for conflicts* on page 47
- *Deploying tailoring changes* on page 49
- *Translating tailored modules* on page 49
- *Adding Get-Services to an existing frameset* on page 54

Peregrine Studio projects

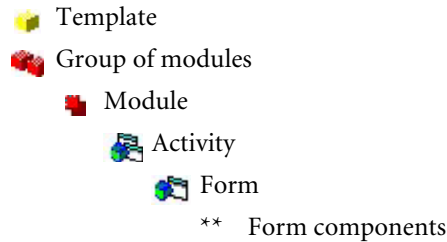
Peregrine Studio saves all the source files for Get-Services as a project. A Studio project consists of several components that are combined during the build process.

Studio component	Description
Get-Services components	The XML files that define the functionality of Get-Services consisting of packages, modules, activities, forms, and form components.
ECMAScripts*	ECMAScripts create and format message objects to the Archway servlet. Get-Services components will use ECMA message objects to display and process data.
Document schema definitions	The XML files that define how the Archway servlet should format the ECMA message objects sent to and received from back-end databases. Get-Services 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-Services.
Stylesheets	The Cascading Style Sheet (CSS) files that define the colors and fonts that will be used in your Get-Services pages.

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



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 same hierarchy:






Project component descriptions

This table lists and describes some of the Studio components. For a complete list of the components that make up a Studio project, refer to Appendix A, beginning on page 199.

Component	Description
Project	<p>The project component:</p> <ul style="list-style-type: none"> ■ 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 Project Explorer tree.
Templates (support files)	<p>The templates component:</p> <ul style="list-style-type: none"> ■ is the container for all the common elements reused throughout the project. ■ appears with a yellow cube icon () in the Project Explorer tree.

Component	Description
Group of modules	<p>The group of modules component:</p> <ul style="list-style-type: none"> ■ is the container for all the supporting files and modules that make up Get-Services. ■ appears with a double red cubes icon () in the Project Explorer tree. ■ does not have any one dedicated graphical representation in the built project.
Module	<p>The module component:</p> <ul style="list-style-type: none"> ■ is a container for the activities and forms that make up Get-Services. ■ appears with a double red box icon () in the Project Explorer tree. ■ appears as a text link on the navigation sidebar and may also appear on the Get-Services Home Menu. <p>Note: The module component is usually where access restrictions are defined. Setting access restrictions limits a module to particular user roles.</p>
Activity	<p>The activity component:</p> <ul style="list-style-type: none"> ■ 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 Project Explorer tree. ■ appears as a text link on the navigation sidebar (Activity Menu).
Form	<p>The form component:</p> <ul style="list-style-type: none"> ■ is where Get-Services user interfaces and displays are defined. ■ appears with a cube and a single window pane icon () in the Project Explorer tree. <p>Note: Typically, the system displays each form component as a page in the main frame.</p>
Form components	<ul style="list-style-type: none"> ■ Form components such as fields, actions, tables, and lookups define the actual user interfaces and displays used in a Get-Services form. ■ Form components appear with a variety of icons in the Project Explorer Tree. ■ Most form components also have a graphical element in a Get-Services form.








Component	Description
Group of scripts	<p>The group of scripts component:</p> <ul style="list-style-type: none"> ■ is a container for all the server-side ECMAScripts used by Get-Services. ■ appears with a document with a yellow border icon () in the Studio Project Explorer tree.
Group of schemas	<p>The group of schemas component:</p> <ul style="list-style-type: none"> ■ is a container for all the document schema definitions that Get-Services uses. ■ appears with a data store and document icon () in the Studio Project Explorer tree.
Group of files	<p>The group of files component:</p> <ul style="list-style-type: none"> ■ 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 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	Saved as	Contains
project 	..\Program Files\Peregrine\Studio\ <i>project\project.adw</i>	<ul style="list-style-type: none"> ■ <package> names ■ Path to package.xml
package 	..\Program Files\Peregrine\Studio\ packages\ <i>package\package.xml</i>	<ul style="list-style-type: none"> ■ <package> name ■ <modules> name ■ <module> names ■ Path to module.xml ■ Schema Names ■ Path to schema.xml ■ Script Names ■ Path to script.xml
modules 	part of ..\Program Files\Peregrine\Studio\ packages\ <i>package\package.xml</i>	<modules> name
module 	..\Program Files\Peregrine\Studio\ packages\ <i>package\modules\module.xml</i>	<ul style="list-style-type: none"> ■ <module> name ■ XML code for <activity>, <form>, and <form> components
schema 	..\Program Files\Peregrine\Studio\ packages\ <i>package\modules\Schemas\schema.xml</i>	XML code for <schema>
script 	..\Program Files\Peregrine\Studio\ packages\ <i>package\modules\ServerScripts\script.xml</i>	XML code for <script>
presentation files 	..\Program Files\Peregrine\Studio\ packages\ <i>package\modules\Presentation</i>	Directory where presentation files can be stored to be included in a Studio build.

Building a project

During the build process, Studio compiles the XML components of your project into a collection of Java Server Pages (JSP). The JSP content will be generated in whatever folders you selected in your Studio Build Settings. Generally, Studio produces one JSP for each form.

XML to JSPs

This table summarizes how the XML content of your project is converted into JSPs during the build process. 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.

Project component	Built as
package	A collection of JSP files, XML files, and string files.
modules	A collection of JSP files, XML files, and string files.
module	A collection of JSP files, XML files, and string files.
activity	A collection of JSP files, XML files, and string files.
form	<i>e_module_activity_form.jsp</i>
form components	as part of <i>e_module_activity_form.jsp</i>

Example

Modules name = *search*

Module name = *employees*

Activity name = *empsearch*

Form name = *search*

XML files saved in project = *search.xml* and *employees.xml*.

JSP created during build = *e_employees_empsearch_search.jsp*.

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.

Important: If you use the **Clean the target folders** option, you will have to redeploy Get-Services in addition to rebuilding the project. It is recommended that you avoid using this option unless you are removing a Get-Services installation.

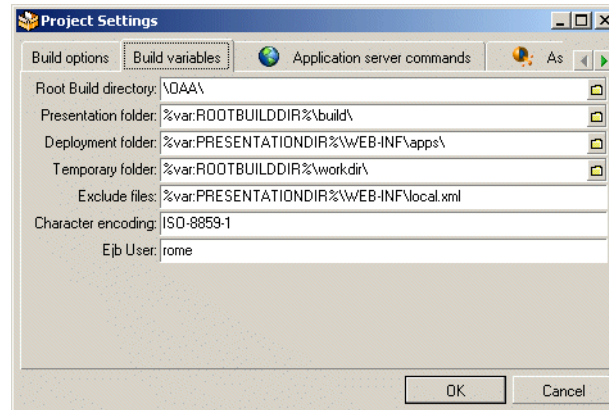
Setting project build settings

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

To set project build settings:

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

The Project Settings window opens.



- 2 Click the **Build Variables** tab.
- 3 Enter or browse to the proper directory for the following settings.
 - Root Build Directory—the root folder for the build.
 - Presentation folder—the folder where you want generated files to be created during a build. Generally, this is the same folder where you create a Web server virtual directory mapping to run Get-Services. Typically, the presentation folder is the **WEB-INF** folder of your application server.
 - Deployment folder—the folder where scripts, schemas, and screen descriptions are located.

Warning: Do not change this option.

- Temporary folder—the folder where Peregrine Studio will generate temporary files used in the build process.
 - Exclude files—a semicolon-separated list of files or directories that you want Peregrine Studio to exclude from removing or rebuilding during a build.
 - Character encoding—Not used. JSP encoding is determined by the character encoding setting on the Settings page of the Admin module.
 - Ejb User—Enterprise Java Beans. The BizDoc database user (rome).
- 4 Click **OK** to save your settings.

Peregrine Studio project packages

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

- System packages. The system packages provided by Peregrine define the out of the box functionality of Get-Services.
- Extension packages. Any packages you create are called *extensions*. Package extensions store all of your additions or modifications. Extensions store only the elements that you have added or changed from the base system package.

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 only 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 packages that ship with Get-Services 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.
 - **Name.** Enter a name for the new package extension. The package extension name cannot contain spaces or special characters.
 - **Dependencies.** Select the existing package or packages that your package extension will be dependent on. Your new package extension must be dependent on at least one existing package. Clear the check boxes of the packages that you do not want your new package to be dependent on.
- 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-Services will now be saved in your new package.

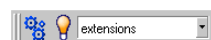
Activating and deactivating packages


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

To activate a package:

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


The Package Activation toolbar is displayed.



- 2 Click the **Package activation** button ().
- 3 Select the checkbox next to the package name or names you want to activate.
- 4 Click OK.

All active packages will be included in the next build.

To deactivate a package:

- 1 Click the **Package activation** button ().
- 2 Clear the check box next to the package name or names you want to deactivate.
- 3 Click OK.

All deactivated packages will be excluded from the next build.

Package dependencies

Each package has a list of dependencies that define what other packages are affected by the current package's changes or additions.

When you create package extensions, you must select the other packages that your new extension will be dependent on. You will be able to make changes or additions to only the packages that are listed in your extension's dependencies. If you try to make changes outside your extension's dependencies, you will produce a 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 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 Information on Selection 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 > Information on Selection**.

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 your Get-Services server. 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.

Translating tailored modules

Out of the box, all Get-It web applications are provided in English. You can order translated versions of Peregrine Studio by purchasing a language pack. Peregrine Studio 4.0 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-Services language packs.

If you tailor your installation of Get-Services, 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-Services, 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 51.

If you do not have a language pack version of Get-Services and you want to create a new translation, refer to the instructions in *Adding new translation strings files* on page 52.

To configure Get-Services to use your new translation, refer to *Configure Get-Services to use new string files* on page 53.

Editing existing translation strings files

You can make edits, additions, and deletions to string files outside of Peregrine Studio using any text editor or standard translation software.

To edit an existing translation string file:

- 1 Open the English string file for your Peregrine Studio project in a text editor or translation program.

You can find all the translation string files in the application server's deployment directories:

- `<application server install>\webapps\oaa\WEB-INF\strings`
- `<application server install>\webapps\oaa\WEB-INF\apps<application group of modules name>`

Note: The English string file will have the ISO-639 two letter abbreviation EN in the file name.

All strings files have a STR file extension.

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

The string file uses the format illustrated below:

```
String_label, "translated string"
```

Where *String_label* is the Peregrine Studio name given to the string, and

Where *translated string* is the actual value of the string to be translated.

For example if you added a new button, you might look for:

```
EMPLOOKUP_EMPLOYEELOOKUP_SEARCH_LABEL, "Search"
```

- 3 Copy the entire line containing the English string.
- 4 Open the string file for the target language in which you want to add a translation.

Note: The string file will use the ISO-639 two letter abbreviation for the language in the file name.

- 5 Paste the copied English string into the target string file. You can paste the string at the end of the string file.

- 6 Change the "*translated string*" portion of the new string to the target language of your translation. For example, to change the string listed above to French, you might enter the following:

```
EMPLOOKUP_EMPLOYEELOOKUP_SEARCH_LABEL, "Recherche"
```

- 7 Save the new string file.

The new translation strings will be available as soon as you stop and restart the application server.

Adding new translation strings files

You can add new string files to provide additional language support to Get-Services. 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-Services.

To edit an existing translation string file:

- 1 Open the English string file for your Peregrine Studio project in a text editor or translation program.

You can find all the translation string files in your application server's installation directory:

- `<application server install>\webapps\oaa\WEB-INF\strings`
- `<application server install>\webapps\oaa\WEB-INF\apps\<application group of modules name>`

Note: The English string file will have the ISO-639 two letter abbreviation EN in the file name.

All strings files have a STR file extension.

- 2 Copy the entire the English string file.
- 3 Create a new string file for the target language in which you want to add a translation.

Note: The string file must use the ISO-639 two letter abbreviation for the language in the file name.

- 4 Paste the copied English string file into the new file.

- 5 Change the "*translated string*" portion of each string to the target language of your translation.
- 6 Save the new string file.
The new translation strings will be available as soon as you stop and restart the application server.

Configure Get-Services to use new string files

- 1 Log in as an administrator (the administrator login page is located at `admin.jsp`).
- 2 Click **Settings**.
- 3 Click the **Common** tab.
- 4 Enter the two letter ISO-639 language code for the languages you want to support in the **Locales** field. The first code entered will be the default language used. The other languages you define will be available in a drop-down list.
- 5 In the **Content type encoding** field, enter the character encoding to be used for the display language. The following table lists some of the common character encoding formats.

Character Encoding	Character Set
ISO-8859-1	U.S. and Western European character sets. This is the default character set used by Studio.
Shift_JIS	Japanese character set
ISO-8859-2	Polish and Czech character set

- 6 Click **Save** at the bottom of the Settings form to save your changes.
- 7 On the Console form, click **Reset Server** to implement your changes.
Users will now be able to select the display language for their session used when they login to the Peregrine OAA Platform.

Adding Get-Services to an existing frameset

You can add Get-Services 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-Services to the parent frameset.

To add Get-Services 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-Services 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.

5 Forms and Form Components

CHAPTER

This chapter provides an introduction to the tailoring Peregrine Studio forms. Topics include:

- *Tailoring forms* on page 56
- *Creating a portal component* on page 72
- *Types of form components* on page 74

Tailoring forms

Each page displayed in Get-Services 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 Appendix A, beginning on page 199.

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-Services 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-Services.
- 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.
 - OR
 - In Peregrine Studio, locate the form in the Project Explorer.
- Step 5** Modify the Get-Services form in Peregrine Studio. Options include:
 - *Changing a form's title* on page 57.
 - *Changing a form's instructions* on page 58.
 - *Changing a form's onload script* on page 59.
 - *Changing a form component's label* on page 60.
 - *Hiding a form component* on page 61.
 - *Changing a form component to read-only* on page 62.
 - *Changing the schema that a form component uses* on page 63.

- *Changing the document field that a form component uses* on page 64.
- *Displaying a form within a frameset* on page 67.
- *Displaying a script variable in a form component* on page 69.

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 Peregrine Studio with several different users with different access rights.

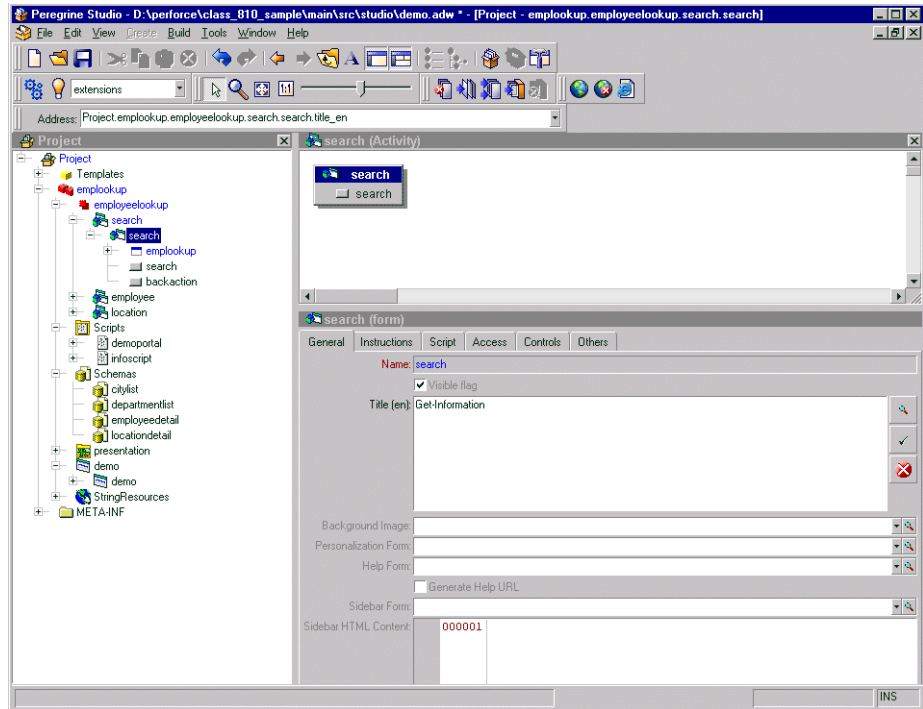
Changing a form's title

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.



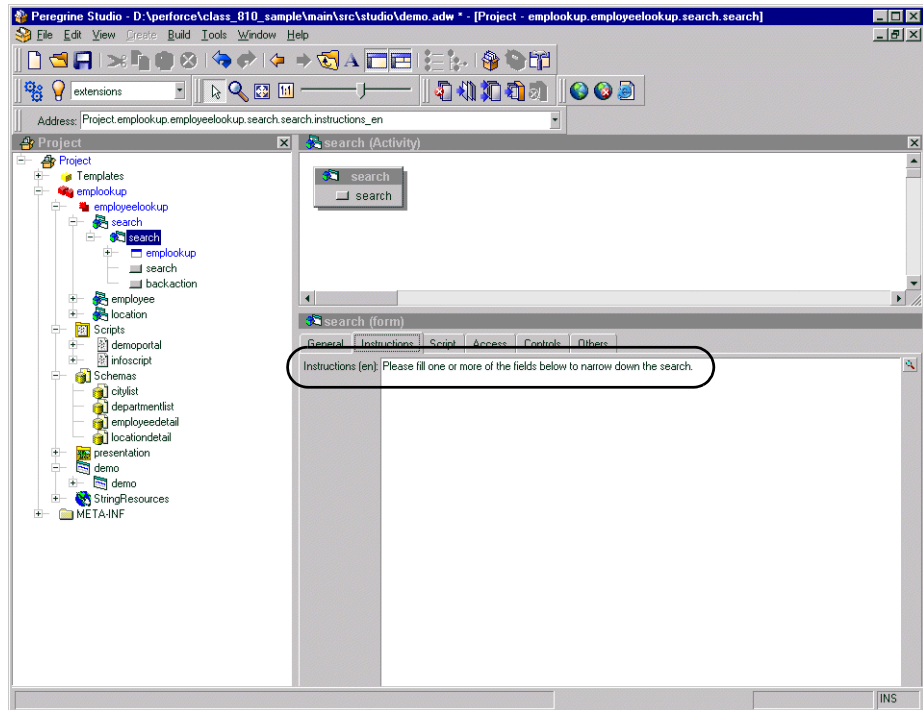
Changing a form's instructions

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 invoked by a form:

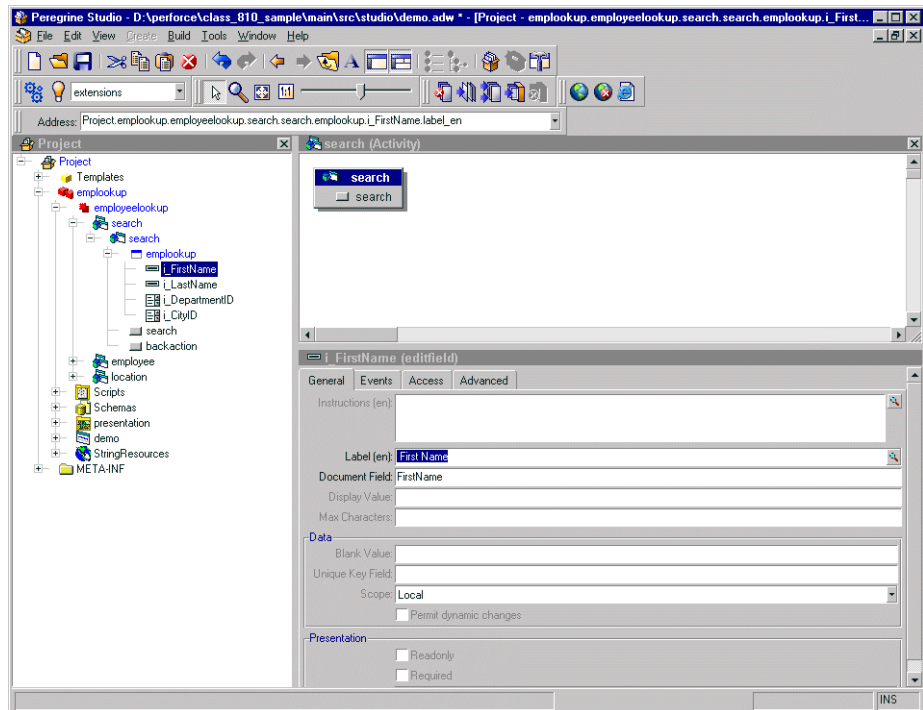
- 1 Select the form in Studio.
- 2 Click the Script tab in the Properties window.
- 3 In the Server Onload Script field, enter or select the script you want to invoke when this form is loaded. You can use the drop-down list to select any of the scripts saved in your project file.
- 4 Save and build your project file.
- 5 Restart your application server.

Changing a form component's label

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

To change a component label (field label):

- 1 Select the form in the Project Explorer.
- 2 On the General tab, select the Label (en) field, enter the new form component label, and press ENTER.
- 3 Save your project.
- 4 Build your project file.



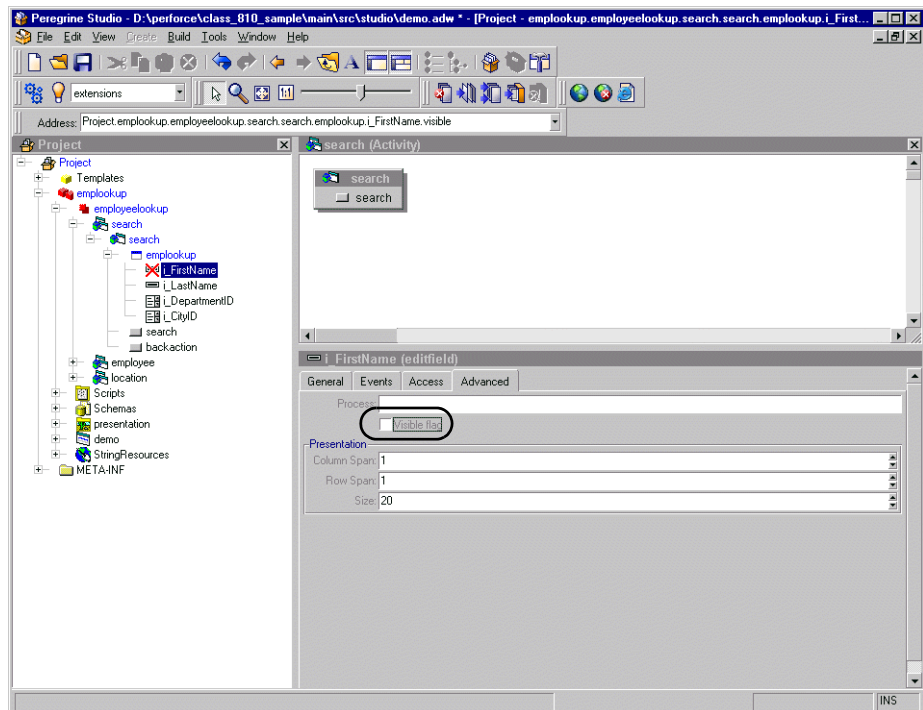
Hiding a form component

All form components have a Visible flag property that hides or displays the component in the Web application interface. If you want to remove a form component from the interface but still have it available in Peregrine Studio, you can toggle the form component's Visible flag to No. This prevents the form component from being part of the next Peregrine Studio build.

Non-visible (and thus non-built) form components are displayed with a red X over the form component icon in the Project Explorer tree.

To hide a form component in the interface:

- 1 Select the form in the Project Explorer.
- 2 On the Advanced tab, clear the Visible flag option.
- 3 Save your project.
- 4 Build your project file.



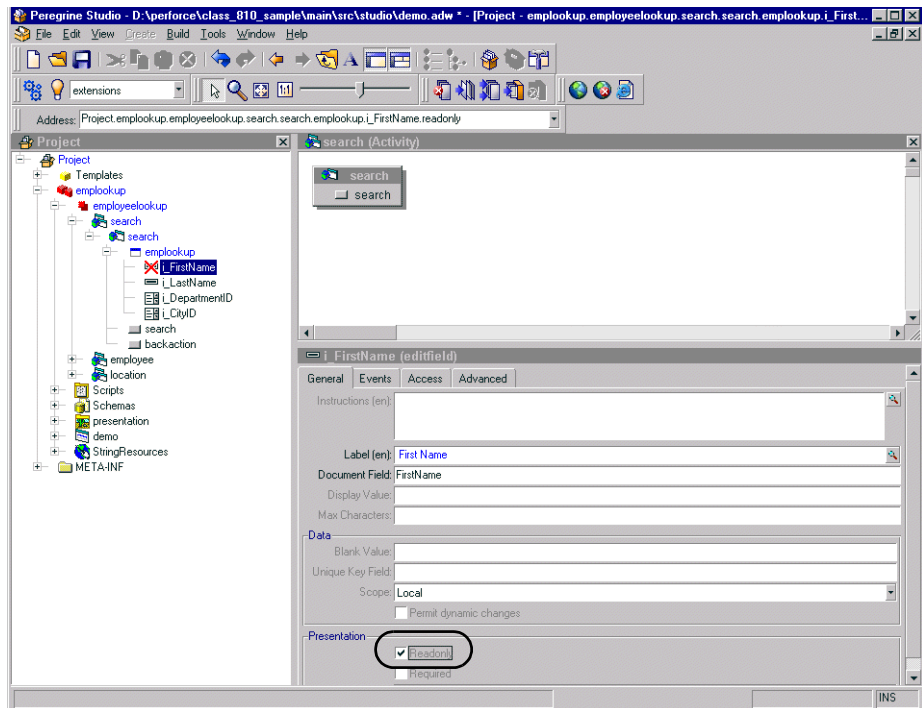
Changing a form component to read-only

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

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

To make a form component read-only:

- 1 Select the form in the Project Explorer.
- 2 On the General tab, select the Readonly check box.
- 3 Save your project.
- 4 Build your project file.

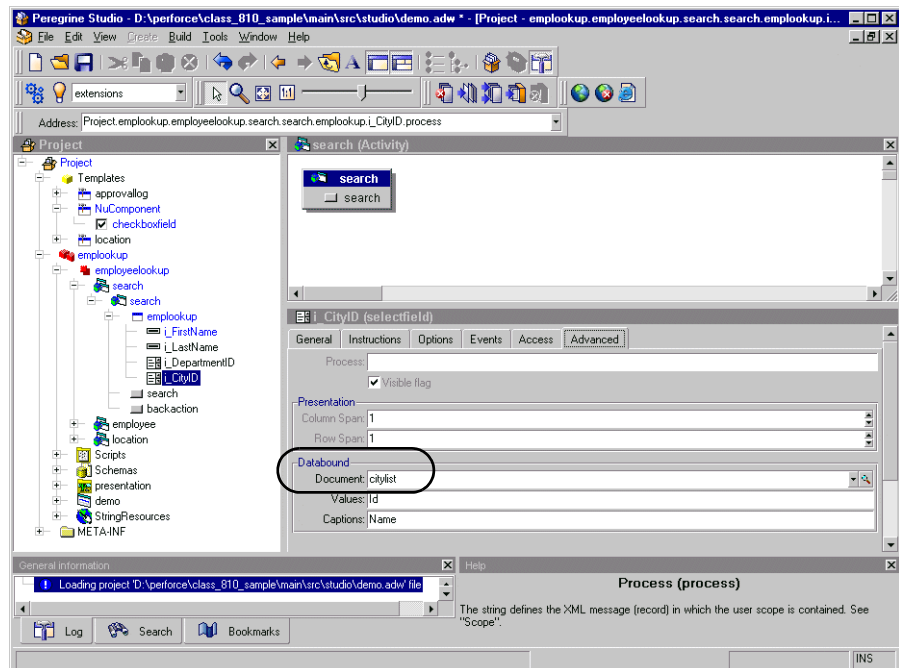


Changing the schema that a form component uses

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

To change the schema that a form component uses:

- 1 Select the form in the Project Explorer.
- 2 Click the Advanced tab.
- 3 In the Databound section, select the **Document** field, and enter or select the name of the schema that you want to use as the source document for this form component.



- 4 Save and build your project file.

Changing the document field that a form component uses

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

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

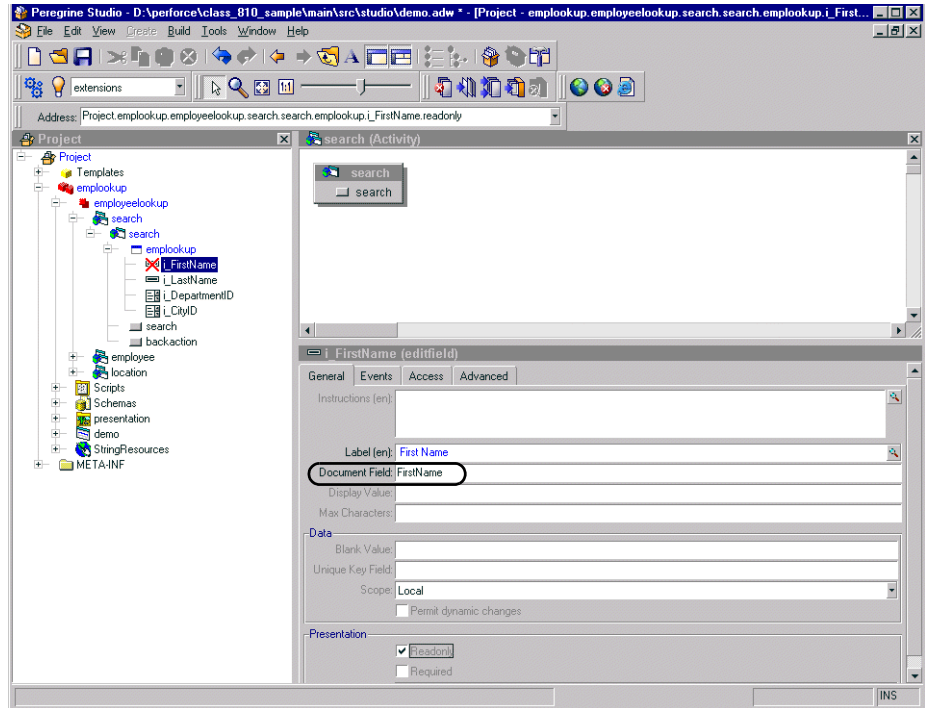
To change the document field that a form component uses:

- 1 Select the form component in Peregrine Studio to display the component's properties.
- 2 In the Document Field field, enter the name of the field in the XML message where this form component's information is stored.

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

- 3 Save your project.

4 Build your project file.



Format of document field name

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

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

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

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

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

- If the Document Field attribute is repeated in the schema, you must specify the nested <document> name or names and the <attribute> name. The <document> name and the <attribute> name must be separated by a slash character (/).
- If the Document Field attribute is part of a nested <document> element, you have the choice of either listing the <attribute> name by itself or specifying the some or all of the path using the syntax of <documents>/<document>/<attribute>. This syntax allows Web application developers to specify as much or as little of the document path as is needed to create a field attribute mapping.

Example

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

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

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

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

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

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

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

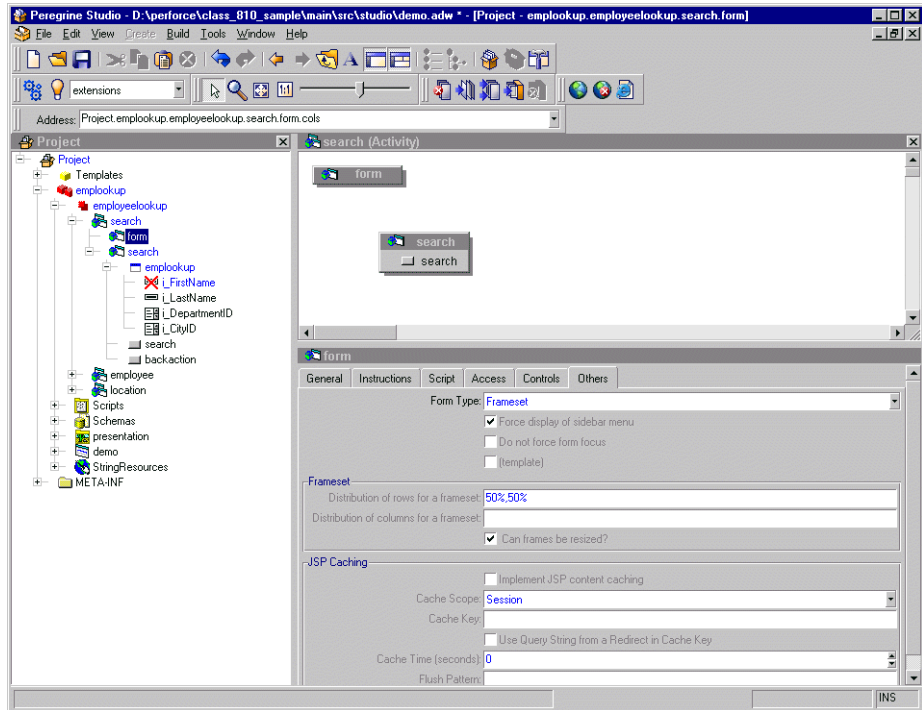
Displaying a form within a frameset

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

To display forms within a frameset:

- 1 Right-click the activity where you want the frameset form to be, point to New, and then click Form.
- 2 Click the Others tab.
- 3 Select Frameset from the Formtype drop-down list box.
- 4 Enter row and column sizes in the Frameset pane.

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



- 5 Create a new form for each frame in the frameset form.
- 6 Create a redirection under the frameset form for each target form in the frameset.
- 7 Save your project.
- 8 Build your project file.

To display the form title within a frameset:

- 1 Open the frameset form's component properties in Studio.
- 2 Create a new server onload script within your project.
- 3 Add the following lines to the script:

```
top.setTitle(TitleText)
```

Where *TitleText* is the title you want to display at the top of the frameset.

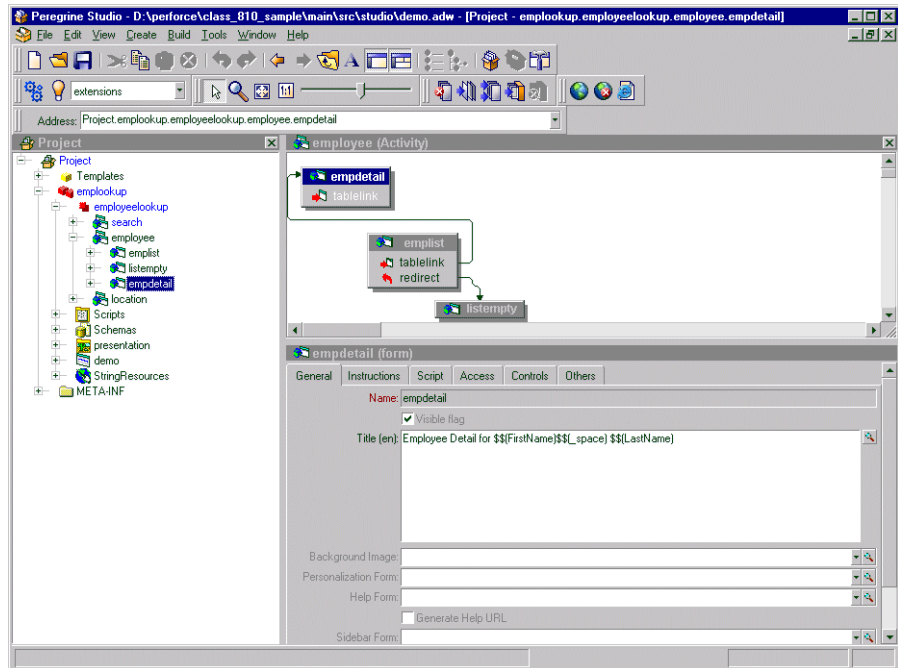
- 4 Open the component properties page for the target form within the frameset.
- 5 Click the Script tab.

- 6 Select the server script you created in step 2.
- 7 Save your project.
- 8 Build your project file.

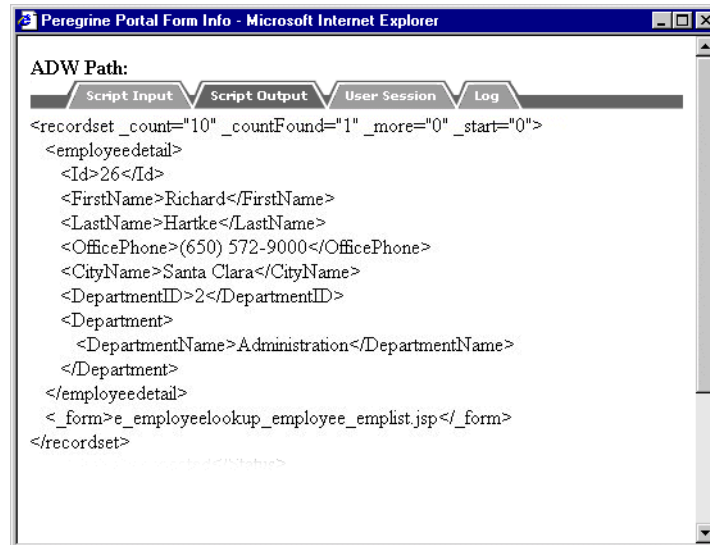
Displaying a script variable in a form component

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.



In the case of a search for an employee named Richard Hartke, the script output might look like the following.

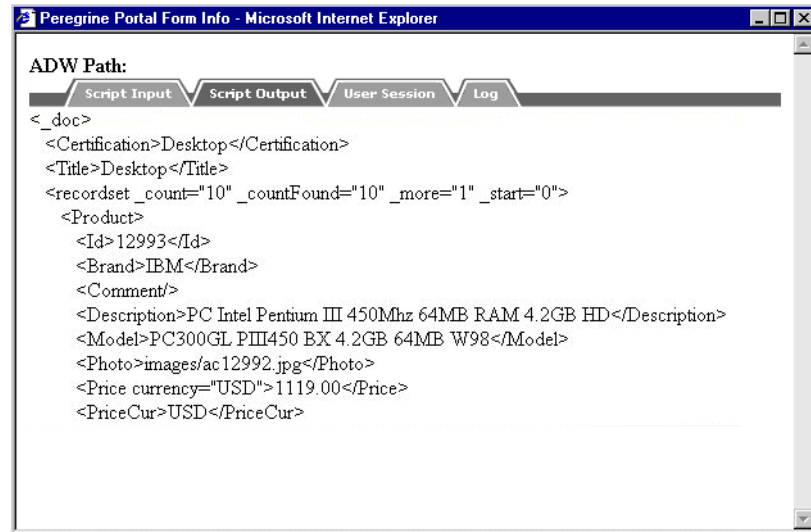


```
ADW Path:
Script Input  Script Output  User Session  Log
<recordset _count="10" _countFound="1" _more="0" _start="0">
  <employee detail>
    <Id>26</Id>
    <FirstName>Richard</FirstName>
    <LastName>Hartke</LastName>
    <OfficePhone>(650) 572-9000</OfficePhone>
    <CityName>Santa Clara</CityName>
    <DepartmentID>2</DepartmentID>
    <Department>
      <DepartmentName>Administration</DepartmentName>
    </Department>
  </employee detail>
  <_form>e_employeelookup_employee_emplist.jsp</_form>
</recordset>
```

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.



The screenshot shows a web browser window titled "Peregrine Portal Form Info - Microsoft Internet Explorer". The main content area displays the "ADW Path" for a product listing. The path is structured as follows:

```
<_doc>
<Certification>Desktop</Certification>
<Title>Desktop</Title>
<recordset _count="10" _countFound="10" _more="1" _start="0">
  <Product>
    <Id>12993</Id>
    <Brand>IBM</Brand>
    <Comment/>
    <Description>PC Intel Pentium III 450Mhz 64MB RAM 4.2GB HD</Description>
    <Model>PC300GL PIII450 BX 4.2GB 64MB W98</Model>
    <Photo>images/ac12992.jpg</Photo>
    <Price currency="USD">1119.00</Price>
    <PriceCur>USD</PriceCur>
```

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-Services packages and source code (included with the Get-Services tailoring kit)
- Peregrine Studio

To create a portal component:

- 1 Open the Get-Services project in Peregrine Studio.
- 2 Right-click the Group of Modules node to which you want to add a portal component and then select **New > Group of Portal Components**.
You do not have to add another Group of Portal Components if one already exists in your project.
- 3 Right-click the Group of Portal Components node from the navigation tree and select **New > Portal Component**.
- 4 Enter the following properties for the portal component:
 - a **Label (en)**. Enter the name you want the portal component to have in the Add/Remove content page.
 - b **Column type**. Select either wide or narrow. This setting determines the size of the portal frame where Peregrine Studio displays the portal component.
 - c **Height of IFRAME**. Enter a height value if you plan to display this portal component from WebSphere Portal Server.
- 5 Right-click on the new portal component and select **New > Contents**.
A standard form page is added.
- 6 Enter any form components, onload scripts, parameters, or access restrictions you want the portal component to have.

Tip: You can use existing Get-Services portal components as a template.

Keep in mind the following considerations:

- Portal components have less space than 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.
- 7 Right-click on the new portal component and select **New > Configure**.
A standard form page is added.
 - 8 On the configure form, add any form components and onload scripts you want to use to configure your portal component.

Important: You can only have one server onload script per portal component that runs from either the contents or the configure forms.

- 9 Save your project file.
- 10 Build your project and deploy your updated JSP 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-Services to display portal components. This setting is available from the Administration page (admin.jsp).

Types of form components

The following sections describe some of the more commonly used form components. For a complete list and description of all Studio components, see Appendix A, beginning on page 199.

Component template containers

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

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

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

To create a component template:

- 1 Right click the **Templates** nodes and click **New > Field Container > Component Template**.
Peregrine Studio adds a new component template node to the Project Explorer Tree.
- 2 Enter the name for the component template.
- 3 Right click the new component template node and use the **New** option to add form components.
- 4 Configure the form components you add to the template component.
Peregrine Studio uses these settings as the default settings of the template component.

- 5 Save and build your Peregrine Studio project.

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

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

To add a component template to a form:

- 1 Right-click the form where you want the component template to be.
- 2 From the New menu, select the template you want to add.

Form components you can add to a component template

- ▶ All except Action and Redirection.

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

Attributes you can set for a component template

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

Fieldsection containers

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

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

To create a fieldsection:

- 1 Right click the form where you want the fieldsection to be.
- 2 Click **New > Field Container > Field section**.

Form components you can add to a fieldsection

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

Attributes you can set for a fieldsection

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

Text edit fields

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

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

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

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

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

To create a text edit field:

- 1 Right-click the container where you want the field to be. This displays the context-sensitive menu.
- 2 Click **New > Field > Text Edit**.

Form components you can add to a text edit field

- ▶ None.

Attributes you can set for a text edit field

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

Selectbox fields

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

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

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

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

- You can only add selectbox fields within a container such as a component template or fieldsection.
- Users cannot add entries to selectbox fields. To implement such functionality, you would need to write a client-side JavaScript to insert any information added into your back-end databases.
- Get-Services uses selectbox fields to constrain user input to a list of predefined items. The selectbox field saves the selected item to a particular field when a user submits the form. The field used to save the information must match a field defined in a document schema.
- If you have a large number of selections for users to choose from you may want to use a lookupfield in place of a selectbox. The advantage of using lookupfields are:
 - they can be personalized
 - they are not loaded into memory until the lookupfield is selected, which reduces the amount of time necessary to render the form.

To create a selectbox field:

- 1 Right click the container where you want the field to be.
- 2 Click New > Field > Selectbox.

Form components you can add to a selectbox field

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

Attribute categories you can set for a selectbox field

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

Databound attributes

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

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

Hidden data fields

A hidden data field stores form information without displaying it to the user. Get-Services 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 to a hidden data field

- ▶ None.

Attributes you can set for a hidden data field

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

Redirections

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

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

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

To create a redirection:

- 1 Right-click the form where you want the redirection to be. The context-sensitive menu is displayed.
- 2 Click **New > Redirection**.

Form components you can add to a redirection

- ▶ None.

Attribute categories you can set for a redirection

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

Redirection attributes

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

- **Condition.** Enter the message generated by your server script that activates the redirection to another form. If there is no condition, the redirection will activate every time the page is loaded.
- **Target form.** Enter the full Peregrine Studio path to the form where the user should be redirected.

Simple table

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

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

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

To create a simple table:

- 1 Right-click the form where you want the table to be.
- 2 Click **New > Table > Simple Table**.

Form components you can add to a simple table

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

Attributes you can set for a simple table

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

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

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

Table links

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

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

To create a table link:

- 1 Right-click the table where you want the table link to be.
- 2 Click **New > Link > Table Link**.

Form components you can add to a table link

- ▶ None.

Attributes you can set for a simple table

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

Table link attributes

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

- Document field. Enter the field that describes what information should be passed when a table link is submitted. The Document Field attribute should match the attribute name of an item in your schema. The attribute is typically set to the Id schema attribute.
- Target form. Enter the full Peregrine Studio path to the form where the user should be redirected when they click on a table row.

Text columns

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

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

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

To create a text column:

- 1 Right click the table where you want the text column to be.
- 2 Click **New > Text Column**.

Form components you can add to a text column

- ▶ None.

Attributes you can set for a text column

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

Text column attributes

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

- Document Field. Enter the field that describes what information should be displayed in the text column. The Document Field attribute should match the attribute name of an item in your schema.
- Label (en). Enter the label you want displayed in the first row of the table as the column heading. If you are using dynamic headers and columns, you will want to leave this attribute blank.

Actions

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

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

To create an action:

- 1 Right-click the form where you want the action to be.
- 2 Click **New > Action** and then click the action type you want to add.

Form components you can add to an action

- ▶ None.

Attributes you can set for an action

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

Action attributes

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

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

6 Adding Personalization Functionality

CHAPTER

This chapter covers the following topics:

- *Supporting personalization* on page 86
- *Personalizing with DocExplorers* on page 89
- *Using the personalization interface* on page 95

Supporting personalization

Personalization of Get-Services 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-Services directly from the Web interface.
- Developers can use Peregrine Studio to add personalization capabilities to their own Get-Services forms by creating new DocExplorers. This functionality can be enabled only by using Peregrine Studio.

To add Personalization capabilities to Get-Services, you must have these components:

- A ServiceCenter back-end database. Personalization requires you to store each user's login rights and personalization changes in a back-end database.
- A user account with personalization rights enabled. A user's login profile determines the level of personalization rights Get-Services 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-Services 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.

Activating personalization

You can grant access to personalization features in one of two ways:

- Grant all users personalization rights with the end-user personalization administrative setting
- Grant specific users personalization rights by adding a security capability to their user profile.

You can globally define end-user access to personalization by selecting one of three options from the End User Personalization options on the **Administration > Settings** page. The end user personalization setting can have one of three values:

- **Enabled.** This global setting enables end-users to personalize any DocExplorer activity that they have rights to see. End-users can add or remove any field listed in the schema used by the DocExplorer. However, only end-users with a `getit.personalization.admin` (or equivalent) security capability will be able to use the advanced explore options.
- **Limited.** This global setting enables end-users to personalize any DocExplorer activity that they have rights to see. Unless they have another security capability with greater rights, end-users can add or remove only the fields that an administrator has revealed. This setting also prevents end-users from changing read-only fields to editable fields.
- **Disabled.** This setting globally turns off all personalization except to users who the administrator has explicitly granted a personalization security capability. All other end-users do not see the personalization wrench icon, and only see the fields that an administrator has configured.

You can individually grant users personalization rights by adding a capability word to the user profile stored in the Get-Services back-end database. The following personalization security capabilities are available:

- `getit.personalization.limited` – User can only personalize features that have been exposed by a user with greater personalization rights.
- `getit.personalization.default` – User can change the layout and add or remove fields from the Get-Services interface.
- `getit.personalization.admin` – User can do all that default security capability can do plus can set personalization options and save personalization changes as the default layout. The admin security capability also grants the following rights:
 - **Document Creation.** Specify which security capabilities can create new records in the back-end database.

- Document Update. Specify which security capabilities can change and submit records to the back-end database.
- Document Deletion. Specify which security capabilities can delete records from the back-end database.
- Save. Enables users to save their personalization changes as the default view that other users see.

Tip: You can use the Save option to propagate personalization changes from administrative users to all other users.

After you add a DocExplorer to a Peregrine Studio project, you can add or remove fields from the personalization Web interface.

Making a schema visible to portal components

The Business View Authoring (BVA) tools – Document List and My Menu – use public schemas to determine what back-end database fields and tables users can see. The Business View Authoring tools can only see the fields and tables that you define in public schemas.

To make a schema visible to portal components:

- 1 Login in to the server where you have installed Get-Services.
- 2 Open Windows Explorer and navigate to your Get-Services apps folder. For example:

```
C:\Program Files\Peregrine\Common\Tomcat4\webapps\oaa\WEB-INF
\apps
```

Each module of your Peregrine Studio project has its own folder of schemas.

- 3 Navigate to the folder name matching the module for which you want to enable public schemas. For example:

```
incidentmgt
```

- 4 Create a text file called `publicSchemas.xml` in this folder.
- 5 Add the following entries to `publicSchemas.xml`:

```
<schemas>
  <document name="Schema Name" label="Label to appear in BVA"/>
  ...
</schemas>
```

Add one `<document>` element for each schema that you want to make available to the Business View Authoring tools.

For the name attribute, enter the file name of the schema as it is listed in Peregrine Studio.

For the label attribute, enter any text that you want to use to describe the schema. This text will appear as a description in the BVA interfaces.

- 6 Save the text file.
- 7 Repeat steps step 3 to step 6 for each module that is in your Peregrine Studio project.

Personalizing with DocExplorers

DocExplorers allow end users a means to create and customize searches of Get-Services data. From the end-user perspective, a DocExplorer is a special activity that allows someone to personalize part of the interface. The user's profile determines the Personalization rights granted.

From the developer's perspective, a DocExplorer is a template activity that allows for the rapid development of Get-Services changes without the need to rebuild a 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.

DocExplorer forms and functions

All DocExplorers provide the following forms and functionality:

- A search form for defining search criteria.
- A list form for displaying search results.
- A details form for displaying detailed information about search results.

If you grant end-users administrative rights, they can also use Personalization for the following actions:

- Add a new record to the database.
- Update existing records in the database.
- Delete existing records from the database.

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-Services or create your own schema entries. For more information on schemas refer to *Document Schema Definitions* on page 129.

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 sc for the ServiceCenter adapter for Get-Services. This is a required parameter.

Replace <SUBTYPE_INSTANCE> with the personalization form subtype you want to invoke or leave blank to use no subtype. See x for more information on personalization form subtypes. This is an optional parameter.

Important: Do not change the target form of the redirect action. This action must go to docExplorer.default.start.

- 9 Save your project.
- 10 Click the **Differential build of project** button to rebuild your project.

Personalizing a DocExplorer reference

After you have added a DocExplorer Reference, you can make changes to this activity directly from the Get-Services Web interface.

To personalize DocExplorer pages:

- 1 Log in to Get-Services.
- 2 Click the activity name for your Document Explorer from the navigation sidebar. By default, the Document Explorer will be called DocExplorer.

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

- 3 Click the wrench icon on the upper right of the interface.
- 4 Make your changes to the search form, and then click **Save**.
Your personalized search form is displayed.
- 5 Click **Search** to display the results list form.
- 6 Click the wrench icon from the upper right of the interface.
- 7 Make your changes to the list form, and then click **Save**.
- 8 Click on any of the results displayed in your personalized list form to go to the detail form.
- 9 Click the wrench icon from the upper right of the interface.
- 10 Make your changes to the detail form, and then click **Save**.

- 11 If you have user rights to create documents, click the activity name for your Document Explorer from the navigation sidebar to return to the search form.
- 12 Click **Create** to display the create form.
- 13 Click the wrench icon from the upper right of the interface. Make your changes to the create form, and then click **Save**.

Adding Personalization to lookup fields

You can create automatically-generated lookup fields using Personalization. These personalization features reduce the number of forms and configuration necessary to create a pop-up window with lookup information. You can use Personalization features to configure two types of lookup fields:


- **Field Lookup**—use this lookup to select one particular field from your schema. For example, you might want to select just the Name field from your Employee schema. See *Field lookup* on page 92.
- **Nested Document Lookup**—use this lookup to select one or more fields that are nested under a subdocument in your schema. For example, you could lookup the Location subdocument from your Employee schema to update several fields such as address, state, zip, and country. See *Nested document lookup* on page 93.

Note: When you select an entry from a Nested Document Lookup, all the fields used by the lookup schema are returned. Any other form components that use these fields will be automatically updated. This allows users to quickly change multiple fields on a form.

Field lookup

To create a field lookup:

- 1 Right click the form to which you want to add the lookup.
- 2 Go to **New > Field > Lookup**.
- 3 Enter the following settings for the Data attributes:
 - **Display Field**—the name field that you want to be displayed in the Web application form when a user makes a selection from the lookup field. If you do not enter a value for this parameter it defaults to the Document Field parameter described below.


- Document Field—the name of the key field used to uniquely identify each individual document record. The value of this field is used to lookup the document field defined in step 3 below. This value will also be posted to the onload script when a particular lookup entry is selected.
- 4 Enter settings for the following DocExplorer Adapter attributes:
 - Adapter—the name of the database adapter where your lookup information is stored.
 - Document Path—the name of the schema and *field* name that you want to lookup and enter into the Web application form. The naming convention used with this parameter is *schema name.field name* with a period (.) between them. For example, the entry *employee.name* will lookup the name field from the *employee* schema.
 - 5 Enter the following setting for the Link Parameters attribute:
 - Target Form—enter `docExplorer.fieldlookup.start` as the form name. This value enables personalization if the end-user has sufficient personalization rights.
 - 6 Click the **Differential build of project** button to rebuild your project.
 - 7 Log in to Get-Services, browse to the updated form, and click the magnifying glass lookup icon () to display a pop-up lookup form.

The lookup field displays a list of values that match the Document Path you entered in step 3 above.
 - 8 If you want to change the field used for the lookup, click the **Personalize this page** link and select the new field you want to use.

Nested document lookup

To create a nested document lookup:

- 1 Right click the form to which you want to add the lookup.
- 2 Go to New > Field > Lookup.
- 3 Enter the following settings for the Data attribute:
 - Display Field—the name field that you want to be displayed in the Web application form when a user makes a selection from the lookup field. If you do not enter a value for this parameter it defaults to the Document Field parameter described below.

- Document Field—the name of the key field used to uniquely identify each individual document record. The value of this field is used to lookup all other document fields of the subdocument. This value is posted to the onload script when a particular lookup entry is selected.
- 4 Enter settings for the following for the DocExplorer Adapter attributes:
 - Adapter—the name of the database adapter where your lookup information is stored.
 - Document Path—the name of the schema and *subdocument* name that defines the subdocument you want to lookup. The naming convention used here is *schema name.subdocument name* with a period (.) between them. For example, the entry *employee.location* will lookup the location subdocument from the employee schema.
 - 5 Enter the following setting for the Link Parameters attribute:
 - Target Form. Enter `docExplorer.documentlookup.start` as the form name.
 - 6 Click the **Differential build of project** button to rebuild your project.
 - 7 Log in to your Web application, browse to the updated form and click the magnifying glass lookup icon () to display a pop-up lookup form.

The lookup field will display a list of values that match the Document Path you entered in step 3 above.
 - 8 If you want to change the subdocument used for the lookup, click the **Personalize this page** link and select the new subdocument you want to use.

Using the personalization interface

You can personalize any Web application interface that displays a wrench icon in the top right of the interface frame. The wrench icon will appear only in activities where a Personalization form has been defined. The Personalization form determines what options are displayed in the Personalization pop-up window.

When you click on the Personalization icon, a pop-up window opens displaying the current settings for the form you are viewing.

Select the fields you want to display in the search screen for **Ticket** documents. Double click on a field in the right column to edit its attributes.

Document Fields	Current Configuration
Available Fields	Current Configuration
-- Left/Right Split --	-- \$\$\$IDS(incidentmgt.sectionSearchCriteria) --
-- Top/Bottom Split --	Ticket Number
-- Section Title --	Status
Asset Tag	Category
Assignment Group	Opened By
Category	Contact
Cause Code	Assignment Group
Company Id	IT Employee
Contact Name	Priority
Field1	
Field2	

Form Options

Title: \$\$\$IDS(studio.explorerTitleSearch,Ticket)

Instructions: \$\$\$IDS(studio.explorerInstructionsSearch,Ticket)

Explorer Options

Skip search: Skip the search page and execute a default query

Create: Go directly to the create screen by default

Single Detail: Go directly to detail page when search finds exactly one item

Summary: Show a summary page for the document

Restrict operations to the following roles:

Document Creation: _____

Document Deletion: _____

Document Update: _____

Revert to Default Save

All personalization pop-up windows have the format described below.

- Available Fields—shows all the document fields and subdocument collections that can be added to the current form. The name of this column will vary depending on the type of form you are viewing. Studio generates the list of available fields by dynamically reading the schema used by the form. Any items listed between dashes are form components you can use to organize and arrange how document fields are displayed in the form.

- **Current Configuration**—shows all the document fields, subdocument collections, and displays components that have been selected for the current form. The first time a form is personalized, this column will be empty.
- **Form Options**—allows you to define the title and instructions the form uses. Get-Services forms use a string file reference so that the title and instructions can be localized. You can use the existing string file reference or add a new one.
- **Explorer Options**—allows you to define which DocExplorer forms Get-Services displays when a particular event occurs. Only users with `getit.personalization.admin` rights will see this section.
- **Restrict operations to the following roles**—allows you to define which user roles can update, create, or delete records from the back-end database system. Only users with `getit.personalization.admin` rights will see this section.
- **Revert to Default**—removes all personalization entered by the end-user and returns the form to the default state. A default form may still display fields if the administrator or the form's schema has defined any default fields to be displayed.
- **Save**—saves and applies your Personalization changes to the current form.

Note: The first time you browse to a new DocExplorer activity, the form will be blank. Use Personalization to add the fields you want to appear on each form.

Adding an existing field to a personalized form

Users can add any field to a personalized form that you have exposed in the DocExplorer's schema. The user's personalization rights determine the list of fields they see from this schema. See *Activating personalization* on page 87 for more information on personalization rights.

If you want to add a field that is not currently available in the DocExplorer's schema, you must create a schema extension. See *Adding a new field to the Available Fields list* on page 136 for more information on adding a new field.

To add an existing field to a personalized form:

- 1 Select a field from the Available Fields list.
- 2 Click **Add**, and the field will appear in the Current Configuration list.
- 3 Click **Save**.

To arrange the order of fields:

- 1 Select a field from the Current Configuration list.
- 2 Click the up arrow or down arrow to change the field's position in the Current Configuration list.
- 3 Click **Save**.

Removing a field from a personalized form

Users can remove fields from any form that offers personalization. The user's personalization rights determine the list of fields that is available for them to remove. See *Activating personalization* on page 87 for more information on personalization rights.

To remove fields from a form:

- 1 Select a field from the Current Configuration list.
- 2 Click the **X** button to remove the field.
- 3 Click **Save**.

Personalizing a field attribute

Each field in a personalization form has its own set of attributes. The user's personalization rights determine the list of attributes that can be configured. See *Activating personalization* on page 87 for more information on personalization rights.

To configure field attributes:

- 1 Double-click a field from the Current Configuration list. A new Personalization pop-up window is displayed.
- 2 Enter the new field attributes:
 - Label—the name to be used as the field label. This name appears next to the field in the interface.
 - Readonly—enter *true* if you do not want users updating information displayed in the field.
 - Required—enter *true* if this field must have a value before a form can be submitted.
- 3 Click Save.

7 Scripting

CHAPTER

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

This chapter covers the following topics:

- *How scripts are used* on page 100
- *Testing Scripts* on page 109
- *Common Message Operations* on page 113
- *Using ECMAScript in an Object Oriented manner* on page 116
- *Sample Scripts* on page 121
- *References* on page 128

How scripts are used

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

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

Types of Scripts

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

- **Server-side scripting**—Server-side scripts run from a Web server. Server-side scripts have access to both user-submitted form data and any data generated by a back-end system. The output of server-side scripts can be returned to both a back-end system and the remote browser. All Get-Services server-side scripts are written in ECMAScript. An example of server-side scripting would be querying a back-end system for the list of open tickets submitted by a user.
- **Client-side scripting**—Client-side scripting runs from a JavaScript-capable browser. Client-side scripts have access to user data before it is submitted to a Web server and any back-end data that was uploaded with the current Web page. The output of client-side scripts can be used only by the client browser. All Get-Services client-side scripts are written in JavaScript. An example of using client-side scripting would be formatting a user's contact information using locale preferences.

Where Scripts are Stored

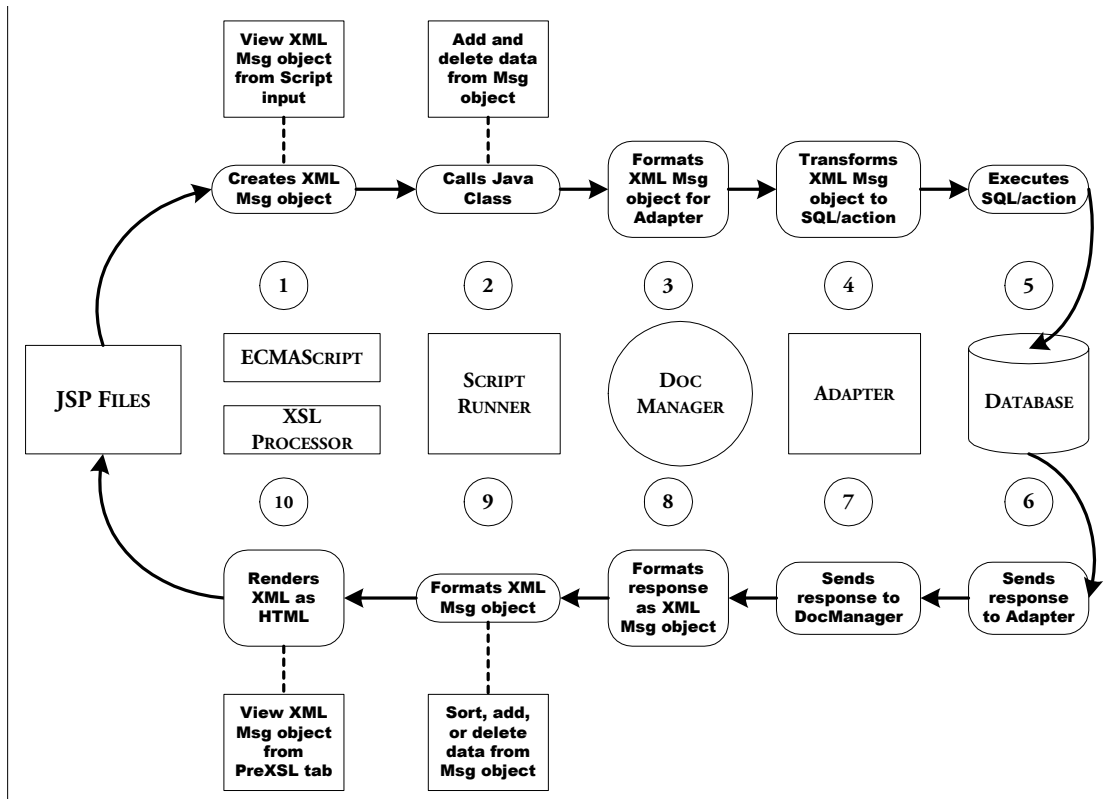
The following table describes how you can include both types of scripting into your projects.

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

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

How Scripts are Used

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



Forms—Server Side

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

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

- `sendDocQuery`—sends an SQL or XML document query to the back-end database. Archway queries the record using the table and field information supplied by the schema. The database then returns the results of the query as an XML document formatted as defined in the schema.
- `sendDocInsert`—sends an XML document to the back-end database that describes a new record. Archway creates the new record in the database using the table and field information supplied by the schema.
- `sendDocUpdate`—sends an XML document to the back-end database that describes an update to an existing database record. Archway updates the record using the table and field information supplied by the schema.
- `sendDocDelete`—sends an XML document to the back-end database that describes a record in the database to be deleted. Archway deletes the record using the table and field information supplied by the schema.

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

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

- For *adapter name*, enter the name for the back-end database adapter. The adapter listed here will use the ODBC connection that you have defined in the `achway.ini` file. For most applications, the adapter will be a two letter name.
- For *schema name*, enter the name defined in the `<document name="schema name">` element of the schema file.
- For the *input msg*, enter the variable name of a message that OAA uses to store input parameters for the ECMAScript function. The default input message is the `msg` object that is defined in all onload functions. The input message is the XML message containing the HTML page parameters.
- For *message object*, enter a variable name of a message object containing a schema name and any input parameters.

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

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

Client Side

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

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

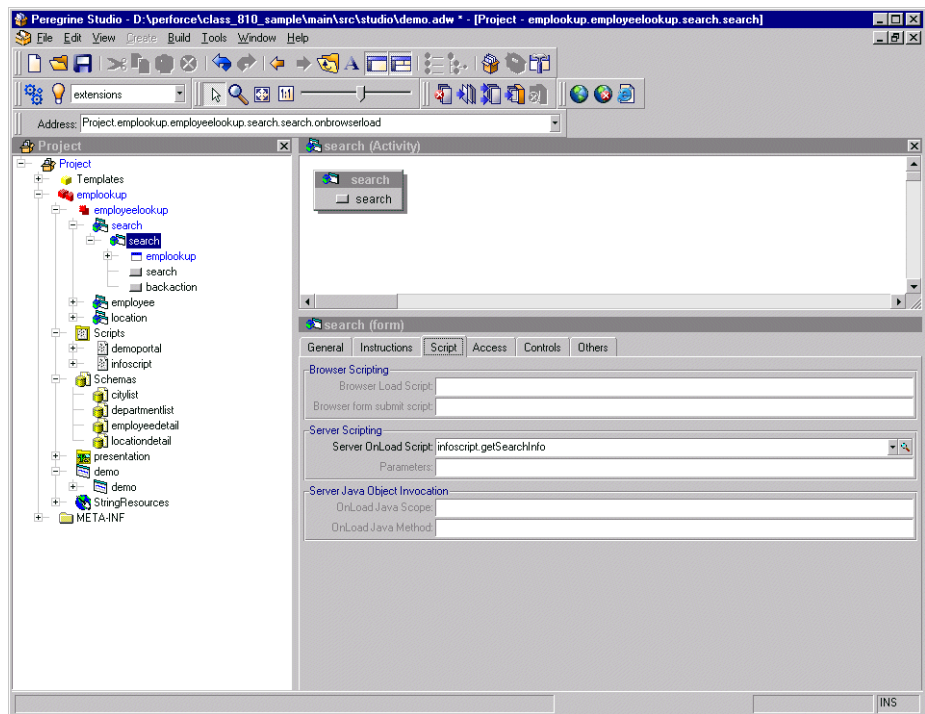
Editing an existing script


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

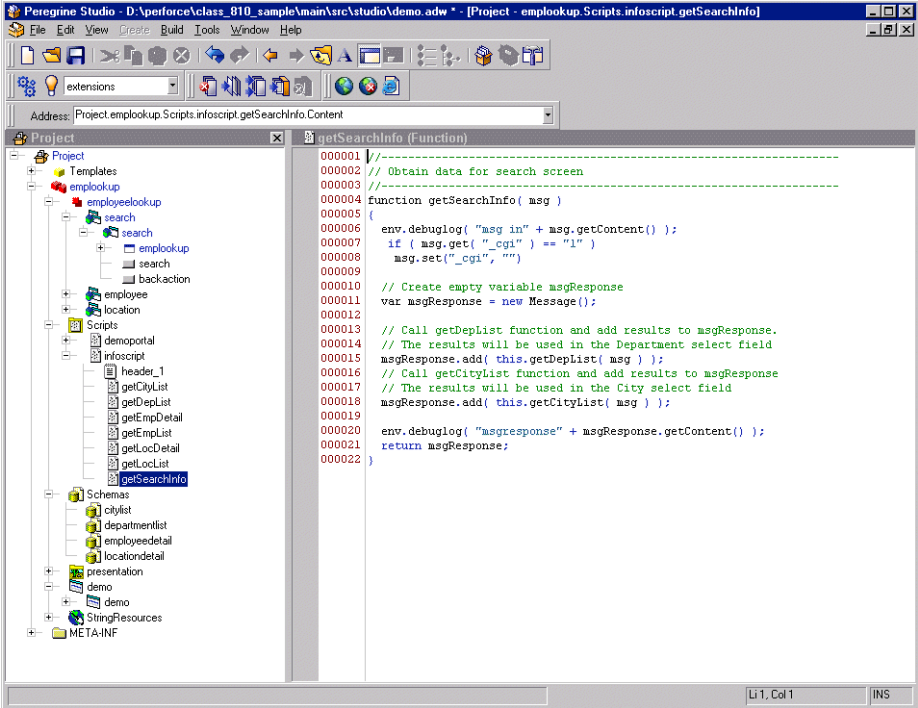
Important: You may lose changes that you make to existing scripts when you next upgrade.

To edit an existing script:

- 1 Select the form in the Project Explorer.
- 2 Click the Script tab in the Properties window.



- In the Server Onload Script field, click the magnifying glass button () to view the script in the Peregrine Studio text editor.



The screenshot shows the Peregrine Studio interface. On the left is a project tree with folders like Templates, employlookup, search, employee, location, Scripts, Schemas, presentation, demo, StringResources, and META-INF. The 'Scripts' folder is expanded, showing various script files, with 'getSearchInfo' selected. On the right, the text editor displays the following JavaScript code:

```

000001 |-----
000002 | Obtain data for search screen
000003 |-----
000004 function getSearchInfo( msg )
000005
000006   env.debuglog( "msg in" + msg.getContent() );
000007   if ( msg.get( "cgi" ) == "1" )
000008     msg.set( "cgi", "" );
000009
000010   // Create empty variable msgResponse
000011   var msgResponse = new Message();
000012
000013   // Call getDepList function and add results to msgResponse.
000014   // The results will be used in the Department select field
000015   msgResponse.add( this.getDepList( msg ) );
000016   // Call getCityList function and add results to msgResponse
000017   // The results will be used in the City select field
000018   msgResponse.add( this.getCityList( msg ) );
000019
000020   env.debuglog( "msgresponse" + msgResponse.getContent() );
000021   return msgResponse;
000022 }

```

- Make any changes to the script in the text editor.
- Save your project.
- Build your project file.
- Restart your application server or set the **File Change Monitor** option from the Administration page.

The script update is loaded into Get-Services.

Adding a custom script

You can add custom scripts to your Peregrine Studio project for use by forms, schemas, and form components.

To add a custom script:

- 1 Determine what kind of script you want to create.

You can create the following types of script:

- **Form onload script.** These are scripts run to gather data for non-DocExplorer forms. Peregrine Studio stores form on-load scripts underneath the first Group of Scripts node (Typically called **Scripts** or **ServerScripts**).
- **Preexplorer.** These are scripts run to manipulate the XML document that gets rendered in the Get-Services 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-Services.

Testing Scripts

Get-Services offers two means of testing your ECMAScript:

- Rhino JavaScript Debugger
- URL Queries

Rhino JavaScript Debugger

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

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

To enable the Rhino JavaScript debugger:

- 1 Login to the Get-Services administration page.
- 2 Click **Settings > Logging** tab.
- 3 For the Debug script option, select **Yes**.
- 4 Click **Save** to store your changes.
- 5 Login to the Get-Services server.
- 6 Browse to the Get-Services deployment directory. By default this directory has the following path:


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

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

For *<context>*, enter the path where you deployed the Get-Services files. For example, `webapps\oaa`.
- 7 Using any text editor, open the file `local.xml`.
- 8 Add the following line anywhere between the `<settings>` elements:


```
<showDebugger>true</showDebugger>
```
- 9 Save the file.

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

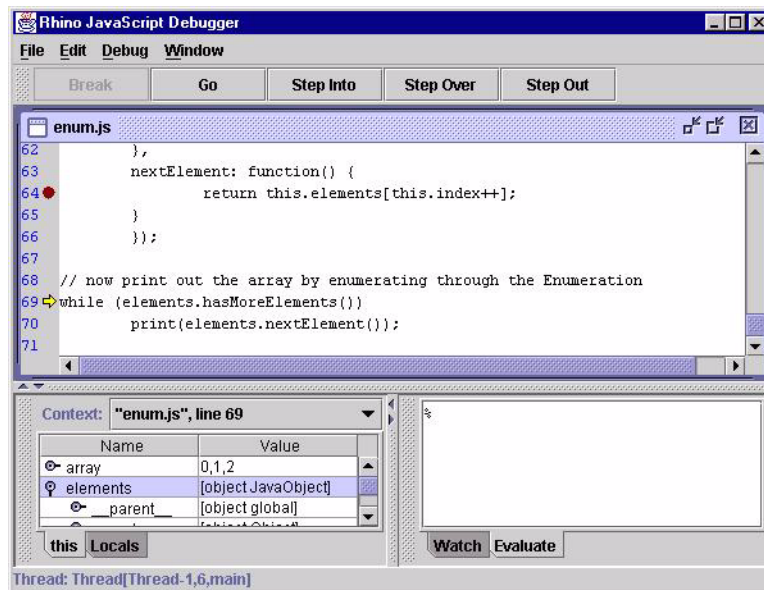
<application server>\<context>\WEB-INF\lib

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

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

- 11 Restart your application server.

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



For more information about the Rhino JavaScript Debugger, see the Mozilla Web site:

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

URL queries

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

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

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

URL Script Queries

Archway URL script queries use the following format:

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

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

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

- For *script name*, enter the name of the script as defined in Studio.
- For *function name*, enter the name of the function used by the script.

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

URL Schema Queries

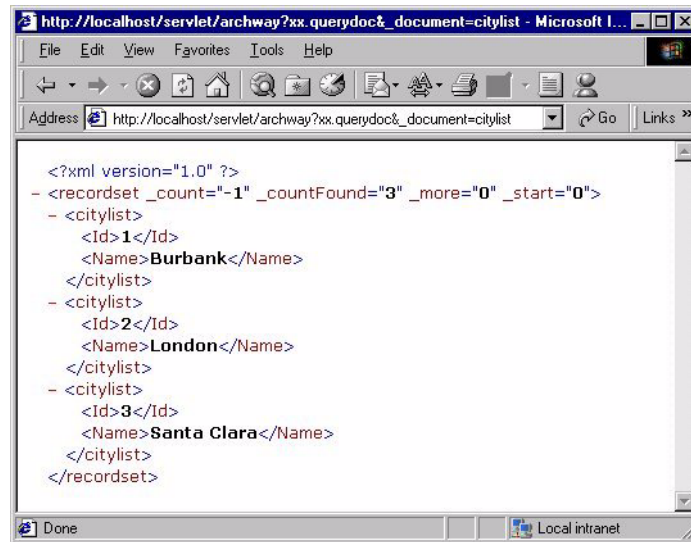
Archway URL schema queries use the following format:

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

- For *adapter name*, enter the name for the back-end database adapter the schema uses. The adapter listed here will use the ODBC connection that you have defined in the Admin module Settings page.
- For *schema name*, enter the name defined in the `<document name="schema name">` element of the schema file.

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

Your script output should be similar to this.



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

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

Common Message Operations

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

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

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

Creates an empty XML message called `Message`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Then, the `msgRequest` uses the subdocument:

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

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

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

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;
while ( i < list.getLength() )
{
  // add function to process records in the list ...
}
```

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

- Log the contents of a particular XML message. This method saves the output of the script to the file `archway.log`. This is another way of debugging your ECMAScript in addition to the *Rhino JavaScript Debugger* on page 109.

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

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

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

Using ECMAScript in an Object Oriented manner

ECMAScript implementation in Get-Services

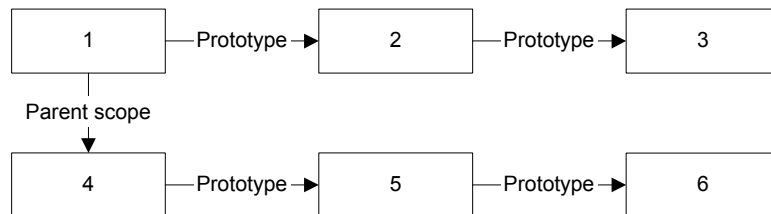
All Scripts defined in Peregrine Studio end up being loaded as one ECMA script object. The functions defined in the Script are the object's methods, and the variables declared outside a function are the object's attributes. This implementation as an object is what enables you to use the **dot** syntax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

Name resolution in ECMAScript

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

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

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



Using the object prototype for object oriented programming

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

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

To use an object prototype:

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

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

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

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

- 2 Create three objects, one for each class:

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

- 3 Try the start method for each of these objects:

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

```
js> myPlane.start()
starting airplane
```

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

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

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

To overwrite a method to extend the parent class method:

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

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

- 2 Create an object for this class:

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

- 3 Call the start method:

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

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

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

To change the start method:

- 1 Create an object for this class:

```
var myFerrari = new sportscar2("Ferrari");
```

- 2 Call the start method:

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

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

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

How to use object orientation for tailoring

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

For example:

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

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

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

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

As long as each object has a unique name, for example the script name, there is no need to store the parent object in a member variable. In that respect, using object orientation in Get-Services 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-Services. If you need help with a client-side scripting, a list of suggested reference materials is provided on page 128.

General Script Samples

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

Selecting a Field from a Schema

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

return msgReturn;
}
```

Input

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

Output

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

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

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

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

Description

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

Calling Other Scripts and Combining the Results

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

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

return msgResponse;
}
```

Input

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

Output

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

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

```

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

Description

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

Form Script Sample

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

Creating an XML Document from a Schema

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

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

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

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

Input

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

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

```

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

```

Output

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

```

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

```

Description

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

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

References

This section contains reference material to help you with scripting.

Sources for Client-side JavaScript

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

JavaDocs for the Main Archway Package

For in-depth information about the Archway servlet and all the functions it supports, refer to the JavaDocs that are installed with Get-Services. The JavaDocs are located in the `\docs\api` folder of your installation. To view the docs, launch the `index.html` file from this folder.

8

Document Schema Definitions

CHAPTER

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

This chapter covers the following topics:

- *Understanding Document Schema Definitions* on page 130
- *How to use schemas* on page 131
- *Schema extensions* on page 132
- *Editing the schema extension files* on page 136
- *Creating custom schemas* on page 148
- *Schema Elements And Attributes* on page 156

Understanding Document Schema Definitions

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

All schemas consist of two types of definitions:

- **Base definitions**—The schema entries that provide a logical mapping between the XML tags generated in a document query to the Get-Services interface are collectively referred to as the schema base definitions. The Archway Document Manager uses the base definitions to generate XML tags based on the elements listed in the schema. The Archway Document Manager converts the name value listed in an <attribute> element into an XML tag of the same name.
- **Derived definitions**—The schema entries that provide a physical mapping between the XML tags generated in a document query to the table and field names in the back-end database are collectively referred to as the schema derived definitions. The Archway Document Manager queries the tables and field names listed in the schema and creates an XML document with the results of the query. The Archway Document Manager converts the table and field values listed in the <document> and <attribute> elements into a SQL query.

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

The base and derived definitions each have their own list of legal elements and attributes. For more information on schema elements and attributes and how to use them, refer to *Schema Elements And Attributes* on page 156.

How to use schemas

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

You can tailor schemas in two ways:

- Create schema extensions. A schema extension is a separate file listing only the changes you make to an existing schema's logical or physical mappings. For example, you could create a schema extension to provide updated physical mappings when you upgrade your back-end database. Creating schema extensions is the preferred method of tailoring schemas as your changes are stored in separate files that can be easily carried over during an upgrade.
- Create new schemas. You can create your own schemas to provide all form components in your project access to the custom logical and physical mappings you create for Get-Services. For example, you could create a new schema to query a collection of custom-created tables and fields that you have added to your back-end database. While you can create new schemas from any text editor without the Get-Services Tailoring Kit, you will need Peregrine Studio to configure and test any server on-load scripts and form components that use your custom-built schemas.

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

Schema extensions

You can create schema extensions to add new *logical* and *physical* mappings to your existing schemas. Schema extensions allow you to save any additional mappings in separate files that preserve the original schema files shipped by Peregrine Systems. This separate file organization ensures that any upgrades will not overwrite your tailoring changes.

When to use schema extensions

Schema extensions generally provide the most benefit when you use them to extend existing DocExplorer schemas. Extending a schema allows you to do the following tailoring tasks without the need to rebuild a project in Peregrine Studio:

- Add new fields to the Available Fields list.
- Hide existing fields from the Available Fields list.
- Change the label that a field displays in the Available Fields list.
- Change the list of forms where a field displays.
- Change the physical mapping of a field.
- Change the type of data a field stores.
- Add subdocuments to the personalization Available Fields list.

For instructions how to perform these schema extension tasks, see *Creating schema extensions* on page 133.

There are some application tailoring tasks where you must use Peregrine Studio to update schema information. These tasks include:

- Call custom scripts from a schema. See *Adding a custom script* on page 107.
- Change the schema used by a non-DocExplorer form component. See *Changing the schema that a form component uses* on page 63.
- Display any new fields that you add to a schema in non-DocExplorer form components such as select fields or tables. See *Changing the document field that a form component uses* on page 64.
- Change the schema used by a DocExplorer. See *Personalizing with DocExplorers* on page 89.
- Add a new schema to your project. See *Creating custom schemas* on page 148.

Creating schema extensions

You can create schema extensions outside of Peregrine Studio using any Text editor. The following procedures outline the steps required to create a schema extension.

To create schema extensions:

- Step 1** Identify the schema that you want to extend. See *Identifying the schema to extend* on page 133.
- Step 2** Locate the schema file on the Get-Services server. See *Locating the schema on the server* on page 134.
- Step 3** Create the schema extension target folders and copy XML files. See *Creating the schema extension target folders and files* on page 135.
- Step 4** Edit the schema extension files to support the features you want. See *Editing the schema extension files* on page 136.

Identifying the schema to extend

You can identify the schema used by a particular form directly from the Get-Services interface. Typically each form uses only one schema, but in some cases a form will use a subdocument that references another schema. The following procedures will help you determine what schema a particular form uses.

To identify the schema used by a particular form:

- 1** Enable Display form information from the **Administration > Settings > Logging** tab page.
The Form information button displays in the banner bar of the Get-Services interface.
- 2** Browse to the form that you want to tailor.
- 3** Click the Display form information button.
The form information window opens.
- 4** Search for one of the following entries on the Script Input tab:
 - `_docExplorerContext`. The last value listed after a slash in this element is the schema name. For example:
`<_docExplorerContext>incident/ticketcontact</_docExplorerContext>`
 uses the `ticketcontact.xml` schema file.

Note: In this example, `ticketcontact.xml` is a subdocument of the primary schema document `incident.xml`. Only DocExplorers will use this *document/subdocument* format.

- `_ctxschema`. The value listed in this element is the schema name. For example:
`<_ctxschema>ticketcontact</_ctxschema>`
 uses the `ticketcontact.xml` schema file.
- `document`. The value listed in this element is the schema name. For example:
`<document>savedRequest</document>`
 uses the `savedRequest.xml` schema file.

If the schema name you find contains an underscore character, for example, `problem_search`, then this schema extends another existing schema. You have the choice of creating a schema extension for either the schema controlling the current form or the parent schema that it extends.

To determine the parent schema name, open the schema, and search for the attribute `extends`. The value of this attribute is the name of the parent schema. For example, the `problem_search` schema has the value `extends="problem"` and therefore extends the `problem` schema.

Tip: If you want to make changes only to one particular form, then create a schema extension of the schema you find listed for the form. If you want to make changes that propagate throughout your Get-Services interface, then create a schema extension of the parent schema listed in the `extends` attribute.

Locating the schema on the server

After you have determined the name of the schema you want to extend, you can find it using your operating system's file search function. The following guidelines are provided to help narrow down your search:

- All schemas files have a `.XML` extension
- All schemas files are stored in the `WEB-INF\apps` folder of your application server's deployment directory. For example:
`C:\Program Files\Peregrine\Common\Tomcat4\webapps\oaa`

Creating the schema extension target folders and files

Schema extensions require two separate files in the same directory where you found the source schema. For example:

```
C:\Program Files\Peregrine\Common\Tomcat4\webapps\oaa\WEB-INF\
apps\incidentmgt\Schemas
```

- Schema extension logical mappings. This file contains the schema base definitions. These definitions determine the logical names and labels used for each field. You must create this file in a sub folder of **Schemas** called **extensions**, and it must have the same name as the schema that it extends. For example:
Schemas\extensions\incident.xml.
- Schema extension physical mappings. This file contains the schema derived definitions. These definitions determine the back-end database tables and fields to which each logical name physically maps. You must create this file in a sub folder of **extensions** that matches the adapter name to your back-end database, and it must have the same name as the schema that it extends. For example:
Schemas\extensions\sc\incident.xml.

To create the schema extension target folders and files:

- 1 Copy the schema XML source file. For example, incident.xml.
- 2 Create two new folders as follows:
 - Create an **extensions** folder in the same directory where you found the source schema. For example:
C:\Program Files\Peregrine\Common\Tomcat4\webapps\oaa\WEB-INF\apps\incidentmgt\Schemas\extensions
 - Create an **<adapter name>** folder in the extension folder.
For *<adapter name>*, enter the abbreviation of the adapter used to connect to your back-end database such as **sc**. For example:
C:\Program Files\Peregrine\Common\Tomcat4\webapps\oaa\WEB-INF\apps\incidentmgt\Schemas\extensions\sc
- 3 Paste a copy of the source schema file in each of the two folders you created.

Editing the schema extension files

The edits that you need to do the schema extension files depend upon what features you are trying to include. The following sections outline what edits you need to perform for each feature.

- *Adding a new field to the Available Fields list* on page 136.
- *Hiding an existing field from the Available Fields list* on page 138.
- *Changing the label a field displays in the Available Fields list* on page 139.
- *Changing the list of forms where a field is visible* on page 140.
- *Changing the physical mapping of a field* on page 142.
- *Changing the type of form component a field uses* on page 143.
- *Adding subdocuments to the Available Fields list* on page 144.

Adding a new field to the Available Fields list

You can add a field to any form that uses personalization. New fields display as options in the personalization Available Fields list.

To add a new field to Available Fields list:

- 1 Open the schema extension file in the extension folder.
This file is for your schema extension logical mappings.
- 2 Delete all the derived definitions listed in the bottom half of the original schema.
The derived definitions section starts after the first `</documents>` element and usually has a comment section describing what back-end databases and versions the derivations apply to.
- 3 In the `<document>` section that remains, add a logical mapping `<attribute>` element for each field you want to add to the list of Available Fields.

You must add each `<attribute>` element between the `<document>` tags:

```

Add new logical
mappings here——— <documents name="base">
                   <document name="schema">
                   <attribute name="Contact" type="string" />
                   </document>
                   </documents>

```

- a Add the required name and type attributes to each `<attribute>` element.

- b** Add any optional attributes you want to use for each `<attribute>` element. Refer to `<attribute>` on page 162 for additional information on the `<attribute>` element.
- 4 Delete any other logical mappings that you will not be updating in the physical mapping schema extension file.

Tip: List only the new logical mappings in your schema extension files. Schema extension entries that duplicate entries in the source schema may reduce your system performance.
 - 5 Save the logical mappings schema extension file.
 - 6 Open the schema extension file in the `<adapter name>` folder. This file is for your schema extension physical mappings.
 - 7 Delete all the base definitions listed in the top half of the original schema. The base definitions section starts with the first `<documents name="base" ...>` element and includes all entries up to the closing `</documents>` element.
 - 8 Find the element `<documents>` that has the name and version attribute values that match the adapter you want to use. For example, `<documents name="sc" version="4">`.

If you cannot find a matching `<documents>` element entry for your adapter, you must create one. See `<documents>` on page 156 for more information on the requirements of a `<documents>` physical mapping.
 - 9 Verify that the `<document>` element beneath your chosen adapter lists the proper table and connection attributes required for your new fields. If the attributes are not what your new fields require, you must edit the attributes. See `<document>` on page 158 for more information on the requirements of a `<document>` physical mapping.
 - 10 Beneath the `<document>` element, add one physical mapping `<attribute>` element for each entry you added in the logical mapping. You must add each `<attribute>` element between the `<document>` tags:

Add new physical mappings here

```
<documents name="sc" version="4.0">
  <document name="schema" table="table1">
    <attribute name="Contact" field="contact_name" />
  </document>
</documents>
```

- a** Add the required name and field attributes for each entry you defined in the logical mapping.

- b** Add any optional attributes you want to use for the physical mapping. See *<attribute>* on page 162 for more information on optional attributes of the *<attribute>* element.
- 11** Delete any other physical mappings that you will not be updating in this schema extension file.
 - Tip:** List only the new physical mappings in your schema extension files. Schema extension entries that duplicate entries in the source schema may reduce your system performance.
- 12** Save the physical mappings schema extension file.

Hiding an existing field from the Available Fields list

You can hide a field from the list of Available Fields in personalized forms. Hidden fields will not be available to any user regardless of user rights.

To hide an existing field from the Available Fields list:

- 1** Open the schema extension file in the extension folder.
This file is for your schema extension logical mappings.
- 2** Delete all the derived definitions listed in the bottom half of the original schema.
The derived definitions section starts after the first *</documents>* element and usually has a comment section describing what back-end databases and versions the derivations apply to.
- 3** Locate the logical mapping for the field you want to remove.
Use the *label* attribute to identify the proper field. For example, if the DocExplorer Available Field you want to remove is called **Contact**, search the *<attribute>* element that has the value *label="Contact"*.
- 4** Add the following four attributes to the *<attribute>* element you want to remove from the DocExplorer Available Fields list:
 - *search="false"*
 - *list="false"*
 - *detail="false"*

■ `create="false"`

```

<documents name="base">
  <document name="schema">
    <attribute name="contact" label="Contact" search="false"
      list="false" detail="false" create="false" />
  </document>
</documents>

```

Add search, list, detail,
and create attributes

These settings tell DocExplorer to hide the field on the search, list, detail, and create forms.

- 5 Delete any other logical mappings that you will not be updating in the physical mapping schema extension file.

Tip: List only the new logical mappings in your schema extension files. Schema extension entries that duplicate entries in the source schema may reduce your system performance.

- 6 Save the logical mappings schema extension file.
- 7 If you will not be making any changes to the physical mappings in this schema, you may delete the schema extension file in the `<adapter name>` folder.

You only need to edit this file if you will define new physical mappings for your DocExplorer fields.

Changing the label a field displays in the Available Fields list

You can change the label that appears in the Available Fields list of personalized forms. Typically, you will only need to add labels to new fields that you have added to a schema.

To change the label a field displays in the Available Fields list:

- 1 Open the schema extension file in the extension folder.
You will define the logical mappings in this file.
- 2 Delete all the derived definitions listed in the bottom half of the original schema.
The derived definitions section starts after the first `</documents>` element and usually has a comment section describing what back-end databases and versions the derivations apply to.
- 3 Locate the logical mapping for the field you want to change.

Use the label attribute to identify the proper field. For example, if the DocExplorer Available Field you want to change is called **Contact**, search the `<attribute>` element that has the value `label="Contact"`.

- 4 Change the label attribute to the new desired value.

Update the label
attribute —————

```
<documents name="base">
  <document name="schema">
    <attribute name="contact" type="string" label="Representative" />
  </document>
</documents>
```

- 5 Delete any other logical mappings that you will not be updating in the physical mapping schema extension file.

Tip: List only the new logical mappings in your schema extension files. Schema extension entries that duplicate entries in the source schema may reduce your system performance.

- 6 Save the logical mappings schema extension file.
- 7 If you will not be making any changes to the physical mappings in this schema, you may delete the schema extension file in the `<adapter name>` folder.

You only need to edit this file if you will define new physical mappings for your DocExplorer fields.

Changing the list of forms where a field is visible

You can determine the list of DocExplorer forms in which a field is visible. By default, a field is visible in all DocExplorer forms.

To change the list of forms where a field is visible:

- 1 Open the schema extension file in the extension folder.
You will define the logical mappings in this file.
- 2 Delete all the derived definitions listed in the bottom half of the original schema.
The derived definitions section starts after the first `</documents>` element and usually has a comment section describing what back-end databases and versions the derivations apply to.
- 3 Locate the logical mapping for the field you want to remove.

Use the label attribute to identify the proper field. For example, if the DocExplorer Available Field you want to remove is called **Contact**, search the <attribute> element that has the value label="Contact".

- 4 Change or add a true value for each DocExplorer form in which you want the field to appear. For example, the following settings will have a field appear in all DocExplorer forms:

- search="true"
- list="true"
- detail="true"
- create="true"

```

Set search, list, detail,
and create attributes ————<documents name="base">
                             <document name="schema">
                               <attribute name="contact" type="string" label="Contact"
                               search="true" list="false" detail="true" create="false" />
                             </document>
                             </documents>

```

- 5 Delete any other logical mappings that you will not be updating in the physical mapping schema extension file.

Tip: List only the new logical mappings in your schema extension files.

Schema extension entries that duplicate entries in the source schema may reduce your system performance.

- 6 Save the logical mappings schema extension file.
- 7 If you will not be making any changes to the physical mappings in this schema, you may delete the schema extension file in the <adapter name> folder.

You only need to edit this file if you will define new physical mappings for your DocExplorer fields.

Changing the physical mapping of a field

You can change the physical mapping that a field uses to point to another back-end database, table, or physical field.

To change the physical mapping of a field:

- 1 Open the schema extension file in the extension folder.

You will define the logical mappings in this file.

- 2 Delete all the derived definitions listed in the bottom half of the original schema.

The derived definitions section starts after the first `</documents>` element and usually has a comment section describing what back-end databases and versions the derivations apply to.

- 3 Locate the logical mapping for the field whose physical mapping you want to change.

Use the `label` attribute to identify the proper field. For example, if the DocExplorer Available Field you want to change is called `Contact`, search the `<attribute>` element that has the value `label="Contact"`.

- 4 Delete any other logical mappings that you will not be updating in the physical mapping schema extension file.

Tip: List only the new logical mappings in your schema extension files. Schema extension entries that duplicate entries in the source schema may reduce your system performance.

- 5 Save the logical mappings schema extension file.

- 6 Open the schema extension file in the `<adapter name>` folder.

This file is for your schema extension physical mappings.

- 7 Delete all the base definitions listed in the top half of the original schema.

The base definitions section starts with the first `<documents name="base" ...>` element and includes all entries up to the first `</documents>` element.

- 8 Find the element `<documents>` that has the name and version attribute values that match the adapter you want to use. For example, `<documents name="sc" version="4">`.

If you cannot find a matching `<documents>` element entry for your adapter, you must create one. See [<documents>](#) on page 156 for more information on the requirements of a `<documents>` physical mapping.

- 9 Verify that the <document> element beneath your chosen adapter lists the proper table and connection attributes required for your new fields.

If the attributes are not what your new fields require, you must edit the attributes. See <document> on page 158 for more information on the requirements of a <document> physical mapping.

- 10 In the <document> section you selected, change the physical mapping <attribute> element to match the new physical mapping you want.

The physical mapping <attribute> elements are between the <document> tags:

Change physical mappings here

```
<documents name="sc" version="4.0">
  <document name="schema" table="table1">
    <attribute name="Contact" field="contact_name" />
  </document>
</documents>
```

- a Change the field attribute to the new physical mapping.
- b Add any optional attributes you want to use for the physical mapping.

Refer to <attribute> on page 162 for more information on optional attributes of the <attribute> element.

- 11 Delete any other physical mappings that you will not be updating in this schema extension file.

Tip: List only the new physical mappings in your schema extension files. Schema extension entries that duplicate entries in the source schema may reduce your system performance.

- 12 Save the physical mappings schema extension file.

Changing the type of form component a field uses

You can change the type of form component a field uses by changing the type attribute value in a schema extension. For a list of all possible types and the form components they use, see <attribute> on page 162.

To change the type of form component a field uses:

- 1 Open the schema extension file in the extension folder.
You will define the logical mappings in this file.
- 2 Delete all the derived definitions listed in the bottom half of the original schema.

The derived definitions section starts after the first `</documents>` element and usually has a comment section describing what back-end databases and versions the derivations apply to.

- 3 Locate the logical mapping for the field you want to change.

Use the `label` attribute to identify the proper field. For example, if the DocExplorer Available Field you want to change is called **Contact**, search the `<attribute>` element that has the value `label="Contact"`.

- 4 Change the type attribute to the new desired value.

```
Update the type attribute——<documents name="base">
  <document name="schema">
    <attribute name="contact" type="string" label="Contact" />
  </document>
</documents>
```

- 5 Delete any other logical mappings that you will not be updating in the physical mapping schema extension file.

Tip: List only the new logical mappings in your schema extension files. Schema extension entries that duplicate entries in the source schema may reduce your system performance.

- 6 Save the logical mappings schema extension file.
- 7 If you will not be making any changes to the physical mappings in this schema, you may delete the schema extension file in the `<adapter name>` folder.

You only need to edit this file if you will define new physical mappings for your DocExplorer fields.

Adding subdocuments to the Available Fields list

You can add a subdocument to add a lookup form component that references information from another schema. Subdocuments have two different formats depending upon the results returned by the schema query. For more information on the schema elements and formats used with subdocuments, see *Subdocuments* on page 169.

To add subdocuments to the Available Fields list:

- 1 Open the schema extension file in the extension folder.
This file is for your schema extension logical mappings.

- 2 Delete all the derived definitions listed in the bottom half of the original schema.

The derived definitions section starts after the first `</documents>` element and usually has a comment section describing what back-end databases and versions the derivations apply to.

- 3 In the `<document>` section that remains, add one of the following sets of elements for each subdocument you want to add to the list of Available Fields:

Element	Condition for use	Subdocument requirements
<code><document></code>	Use if the subdocument query always returns <i>one and only one</i> result for each requested element in the subdocument. For example, a contact should only have one name.	Required attributes <ul style="list-style-type: none"> ■ name Optional attributes <ul style="list-style-type: none"> ■ docname
<code><collection></code>	Use if the subdocument query can return <i>more than one</i> result for each requested element in the subdocument. For example, a contact can have multiple tickets open in his name.	Required attributes <ul style="list-style-type: none"> ■ name Required elements <ul style="list-style-type: none"> ■ <code><document></code>

```

<documents name="base">
  <document name="schema">
    <attribute name="contact" type="string" label="Contact" />
    ...
    <document name="address" docname="external_schema" />
    ...
  </document>
  <collection name="telephone_numbers">
    <document name="telephone_number" />
  </collection>
  ...
</documents>

```

Subdocument with one result – address

Subdocument with multiple results – telephone numbers

- 4 Delete any other logical mappings that you will not be updating in the physical mapping schema extension file.

Tip: List only the new logical mappings in your schema extension files. Schema extension entries that duplicate entries in the source schema may reduce your system performance.

- 5 Save the logical mappings schema extension file.

- 6 Open the schema extension file in the `<adapter name>` folder.
This file is for your schema extension physical mappings.
- 7 Delete all the base definitions listed in the top half of the original schema.
The base definitions section starts with the first `<documents name="base" ...>` element and includes all entries up to the first `</documents>` element.
- 8 Find the element `<documents>` that has the name and version attribute values that match the adapter you want to use. For example, `<documents name="sc" version="4">`.

If you cannot find a matching `<documents>` element entry for your adapter, you must create one. See `<documents>` on page 156 for more information on the requirements of a `<documents>` physical mapping.
- 9 Verify that the `<document>` element beneath your chosen adapter lists the proper table and connection attributes required for your new fields.

If the attributes are not what your fields require, you must edit the attributes. See `<document>` on page 158 for more information on the requirements of a `<document>` physical mapping.

- 10 Beneath the <document> element, add one of the following sets of elements for each logical subdocument that you added:

Element	Condition for use	Subdocument requirements
<document>	Use if the subdocument query always returns <i>one and only one</i> result for each requested element in the subdocument. For example, a contact should only have one name.	Required attributes <ul style="list-style-type: none"> ■ table ■ field ■ joinfield ■ joinvalue Optional attributes <ul style="list-style-type: none"> ■ docname
<collection>	Use if the subdocument query can return <i>more than one</i> result for each requested element in the subdocument. For example, a contact can have multiple tickets open in his name.	Required attributes <ul style="list-style-type: none"> ■ name Required elements <ul style="list-style-type: none"> ■ <document>

```

<documents name="sc" version="4.0">
  <document name="schema" table="table1">
    <attribute name="contact" field="contact_name"/>
    ...
    <document name="address" table="table2" joinfield="addressee"
      joinvalue="id" />
    ...
    <collection name="telephone_numbers">
      <document name="telephone_number" table="table3"
        joinfield="contact" joinvalue="id" />
    </collection>
    ...
  </document>
</documents>

```

Subdocument maps to external table – table2

Subdocument maps to external table – table3

- 11 Delete any other physical mappings that you will not be updating in this schema extension file.

Tip: List only the new physical mappings in your schema extension files. Schema extension entries that duplicate entries in the source schema may reduce your system performance.

- 12 Save the physical mappings schema extension file.

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 give 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 *Schema extensions* on page 132.

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-Services server. The following procedures outline how to create a custom schema.

- Step 1** Create or activate a package extension to save your changes in Peregrine Studio. See *Peregrine Studio project packages* on page 44.
- Step 2** Add a new schema file to your Peregrine Studio project. See *Adding a schema to your Peregrine Studio project* on page 149.
- Step 3** Add logical and physical mappings to your schema file. See *Adding logical and physical mappings to your schema* on page 149.
- Step 4** Configure other project components to use your custom schema. See *Forms and Form Components* on page 55.
- Step 5** Rebuild your Get-Services project. See *Building a project* on page 41.
- Step 6** Deploy your new Get-Services project files. See *Deploying tailoring changes* on page 49.

Adding a schema to your Peregrine Studio project

You can only add a custom schema to a *group of schemas* node. This node will also be a child element of a *group of modules* node, and typically has the name *Schemas*.

To add a schema to your Peregrine Studio project:

- 1 Right-click the group of schemas node to which you want to add a schema. This node will be underneath the group of modules node for Get-Services. If your project contains more than one group of modules, choose the one that has a group of schemas node.
- 2 Point to New, and then click **Raw Schema**.
A new node appears with the name **Schema**.
- 3 Rename your schema using the following conventions.

Schema Naming Conventions

Each custom schema you create should have a unique name to prevent data errors from naming conflicts. Your custom schema name should meet the following criteria:

- The schema name is unique from any other schema name in the Peregrine Studio project.
- The schema name is unique from any attribute name mapping within the schema.

Adding logical and physical mappings to your schema

After you have added a new schema to your Peregrine Studio project, you are ready to add logical and physical mappings. Studio displays the content of your custom schema in a text editor window. You can use the text editor window to review and edit the XML source code of your schema. You can also use any text editor to edit your schema.

Note: If you use an external text editor to edit your custom schema, Peregrine Studio will not pick up the changes until the next time you open the project file.

All schemas must have both a logical and a physical mapping section. The logical mapping section is where you define what names and labels Get-Services 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 150.
- Step 2** Add two `<documents>` elements for the logical mappings. See *Adding logical mapping <documents> elements* on page 150.
- Step 3** Add two `<document>` elements to define the schema name. See *Adding logical mapping <document> elements* on page 151.
- Step 4** Add one `<attribute>` element for each logical mapping you want to create. See *Adding logical mapping <attribute> elements* on page 151.

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-Services uses this attribute value to determine how to render the field in the user interface. For more information about the type attribute, see [<attribute>](#) on page 162.

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

For more information about the attributes available for the <attribute> element, see *<attribute>* on page 162.

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 152.
- Step 2** Add two <document> elements to define the back-end database table name. See *Adding physical mapping <document> elements* on page 153.
- Step 3** Add one <attribute> element for each logical mapping you created. See *Adding physical mapping <attribute> elements* on page 154.

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>
```

Add a second set of <documents> elements here

These elements are the container for the physical mappings.

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

```
<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 *sc*.

- 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="sc" 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="sc">
      <document>
        <document/>
      </documents>

    </schema>
```

Add a second set of
<document> elements
here

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

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

```
<document name="table_name">
```

For *table_name*, enter the SQL name of the table you want to map to. The Archway Document Manager uses this attribute value to query the back-end database table.

- 3 Add any optional attributes to the <document> element that you want to use to connect to the back-end database or to run process scripts.

For more information about the attributes available for the <document> element, see [<document>](#) on page 158.

Adding physical mapping <attribute> elements

- 1 Add one <attribute> element between the physical mapping <document> elements for each logical mapping you created:

```
<attribute />
```

Note: You can use the standard XML self-closing tag syntax <element /> with the <attribute> element. You can also close every <attribute> element with a </attribute> element if you want.

- 2 Add the identical name attribute to each <attribute> element as you defined in the logical mappings:

```
<attribute name="sample" />
```

Each logical mapping <attribute> element must have a matching physical mapping <attribute> element. The Archway Document Manager uses this value to determine which logical name maps to a particular back-end database field.

- 3 Add a field attribute to each <attribute> element:

```
<attribute name="sample" field="field_name" />
```

For *field_name*, enter the SQL name of the field you want to map to. The Archway Document Manager uses this attribute value to query the back-end database field.

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

For more information about the attributes available for the <attribute> element, see [<attribute>](#) on page 162.

Sample schema

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

```

XML namespace -----<?xml version="1.0"?>
                    <schema>

                    <!--=====
                    Logical Mappings: XML elements and data types defined
                    =====>
Logical mappings always
use name="base" -----<documents name="base">
Document name -----<document name="sample">
determines schema name.
This schema is sample.xml
                    <attribute name="Id" type="number">
                    <attribute name="contact" type="string" label="Contact" />
                    </document>
                    </documents>

                    <!--=====
                    Physical Mappings: Logical names mapped to SQL names
                    =====>
Physical mapping lists
adapter name -----<documents name="sc">
Physical mapping uses
same attribute elements ---<document name="sample" table="incidents">
                    <attribute name="Id" field="incident.id" />
                    <attribute name="contact" field="contact.name" />
                    </document/>
                    </documents>

                    </schema>

```

Schema Elements And Attributes

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

<?xml>

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

<schema>

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

<documents>

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

Use in logical mapping

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

Required attributes

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

Optional attributes

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

Logical mappings always use name="base" ———

```
<?xml version="1.0"?>
<schema>
  <documents name="base">
    ...
  </documents>
  ...

```

Use in physical mapping

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

Required attributes

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

Optional attributes

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

```
<?xml version="1.0"?>
<schema>
  ...
  <documents name="sc" version="4">
    ...
  </documents>
  <documents name="sc" version="5">
    ...
  </documents>
  ...

```

You can add a `<documents>` element for each adapter

Each `<documents>` element can describe a different version

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

- If the <documents> element has *no* version attribute, then the Archway Document Manager accepts the physical mappings in this element if it cannot find another matching value.
- If the <documents> element has a version attribute value *greater* than the version number of the back-end database, then the Archway Document Manager ignores the physical mappings in this element.
- If the <documents> element has a version attribute value *less* than the version number of the back-end database, then the Archway Document Manager accepts the physical mappings in this element if it cannot find a higher matching value.
- If the <documents> element has a version attribute value *equal* to the version number of the back-end database, then the Archway Document Manager accepts the physical mappings in this element.

<document>

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

Use in logical mapping

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

Required attributes

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

Optional attributes

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

- **ACLdelete.** This attribute determines the default access control list for DocExplorer forms that use this schema. The value of this attribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see a **Delete** button in DocExplorer forms that use this schema.
- **ACLupdate.** This attribute determines the default access control list for DocExplorer forms that use this schema. The value of this attribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will be able to edit fields in DocExplorer detail forms that use this schema.
- **create.** This attribute determines if a subdocument using this element is visible in DocExplorer *create* forms. The value of this attribute must be either true or false. Set the value to `create="true"` if you want this subdocument to be available on DocExplorer create forms. Set the value to `create="false"` if you want to prevent this subdocument from being available on DocExplorer create forms.
- **detail.** This attribute determines if a subdocument using this element is visible in DocExplorer *detail* forms. The value of this attribute must be either true or false. Set the value to `detail="true"` if you want this subdocument to be available on DocExplorer detail forms. Set the value to `detail="false"` if you want to prevent this subdocument from being available on DocExplorer detail forms.
- **docname.** This attribute defines the external schema that you want the Archway Document Manager to use to create a subdocument. The value of this attribute must match the file name of the schema (*without* the .xml extension) that you want to use for the subdocument. You only need this attribute if you want to create a subdocument using an another schema.
- **label.** This attribute determines what name the schema has in DocExplorer forms that use this schema. The value of this attribute can be any text string. Typically, you will want to set this value to a user-friendly name describing the content of the schema.
- **list.** This attribute determines if a subdocument using this element is visible in DocExplorer *list* forms. The value of this attribute must be either true or false. Set the value to `list="true"` if you want this subdocument to be available on DocExplorer list forms. Set the value to `search="false"` if you want to prevent this subdocument from being available on DocExplorer list forms.

- **loadscript.** This attribute determines what ECMAScript runs when this schema is used in a DocExplorer form. The value of this attribute must be the Peregrine Studio name of the ECMAScript you want to run. You can use this script to load additional data for use by DocExplorer forms. This script uses the same XML message input as the form onload script.
- **preexplorer.** This attribute determines what ECMAScript runs when this schema is used in a DocExplorer form. The value of this attribute must be the Peregrine Studio name of the ECMAScript you want to run. You can use this script to make formatting changes to the XML message rendered by DocExplorer forms.
- **search.** This attribute determines if a subdocument using this element is visible in DocExplorer *search* forms. The value of this attribute must be either true or false. Set the value to `search="true"` if you want this subdocument to be available on DocExplorer search forms. Set the value to `search="false"` if you want to prevent this subdocument from being available on DocExplorer search forms.
- **subtypeprop.** This attribute determines whether this element inherits the attribute properties of the parent `<collection>` element. The value of this attribute must be `inherit` if you use the attribute at all. If you want this element to inherit the attribute properties set the value to `subtypeprop="inherit"`. If you want to specify the the attribute properties for this element, do not include a `subtypeprop` attribute.

Use in physical mapping

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

Required attributes

- **name.** This attribute determines what XML element the Archway Document Manager matches to a back-end database table. The value of this attribute must match the file name of the schema (*without* the .xml extension).
- **table.** This attribute identifies the table in the back-end database that the schema uses. The value of this attribute must be the SQL name of the table you want to use for source data. Each `<document>` element can only have one table attribute. To use data from other tables, you can create subdocuments within your schema.

Optional attributes

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

- **field.** This attribute identifies the field in the back-end database that you want the schema to use for document queries. The value of this attribute must be the SQL name of the field you want to use for the data source. You only need this attribute if you want to create a subdocument within your schema.
- **insert.** This attribute identifies the event name to be sent to ServiceCenter when Get-Services inserts (creates) a new record. The value of this attribute must be the SQL name of the ServiceCenter event.
- **joinfield.** This attribute identifies the field in the back-end database that you want the schema to use to query for additional information in another schema or table. The value of this attribute must be the SQL name of the field you want to use for the source data. You only need this attribute if you want to create a subdocument within your schema. The `joinfield` attribute defines what field will be the selection criteria in a SQL WHERE clause. The SQL equivalent of the `joinfield` is:

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

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

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

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

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

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

- preprocess. This attribute determines what ECMAScript runs *before* the Archway Document Manager connects to the back-end database. The value of this attribute must be the Peregrine Studio name of the ECMAScript you want to run. You can use this script to format the request sent to the back-end database. For example, you can add additional SQL commands or validate that all required fields are listed in the request.
- postprocess. This attribute determines what ECMAScript runs *after* the Archway Document Manager receives a response from the back-end database. The value of this attribute must be the Peregrine Studio name of the ECMAScript you want to run. You can use this script to format the response sent from the back-end database. For example, you can sort the data by a particular criteria or return an error message if no records are found.
- update. This attribute identifies the event name to be sent to ServiceCenter when Get-Services updates an existing record. The value of this attribute must be the SQL name of the ServiceCenter event.

<attribute>

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

Use in logical mapping

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

Required Attributes

- name. This attribute determines the XML tag that the Archway Document Manager generates when it uses the schema. The value of this attribute can be any string value. For example, if you set the value to name="contact" then the Archway Document Manager creates a <contact> XML tag. You must define at least one <attribute> element where the name attribute has the value name="id". This <attribute> element is required to uniquely identify each record returned by a schema query.
- type. This attribute determines what data format the elements uses as well as how Get-Services renders the data in the user interface. The value of this attribute must be one of the following strings:
 - attachment—This element is a path and file name to an attachment. Get-Services renders this element as a collection of attachment controls.

- **boolean**—This element is a true or false string. Get-Services renders this element as a check box.
- **date**—This element is a date listing. Get-Services renders this element as a date edit control that includes a popup calendar.
- **datetime**—This element is a combined date and time listing. Get-Services renders this element as a time edit control.
- **id**—This element is a number that uniquely describes a back-end database record. Get-Services renders this element as a single-line edit field.
- **image**—This element is an image. Get-Services renders this element as an imagefield.
- **link**—This element is a subdocument described elsewhere in the schema. Get-Services renders this element a lookup field.
- **memo**—This element is a text string. Get-Services renders this element a multi-line edit box.
- **money**—This element is a currency amount. Get-Services renders this element a money field that includes a currency selection tool.
- **number**—This element is an integer. Get-Services renders this element an editfield with spinner buttons.
- **preload**—This element is an executable script. Get-Services runs the script listed in this element.
- **string**—This element is text. Get-Services renders this element an editfield.
- **time**—This element is a time listing. Get-Services renders this element as a time edit control.
- **url**—This element is a Web site address. Get-Services renders this element as an HREF link icon.

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

Optional attributes

- **access.** This attribute determines if the element is read-only or editable in DocExplorer forms. The value of this attribute must be either r or null. Set the value to access="r" if you want to make this element read-only. Clear the value or remove the attribute if you want to make the element editable.

- **ACLcreate**. This attribute determines the default access control list for DocExplorer forms that use this element. The value of this attribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see this element in DocExplorer *create* forms that use this schema.
- **ACLdetail**. This attribute determines the default access control list for DocExplorer forms that use this element. The value of this attribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see this element in DocExplorer *detail* forms that use this schema.
- **ACLlist**. This attribute determines the default access control list for DocExplorer forms that use this element. The value of this attribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see this element in DocExplorer *list* forms that use this schema.
- **ACLsearch**. This attribute determines the default access control list for DocExplorer forms that use this element. The value of this attribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see this element in DocExplorer *search* forms that use this schema.
- **create**. This attribute determines if the element is visible in DocExplorer *create* forms. The value of this attribute must be either true or false. Set the value to `create="true"` if you want this field to be available on DocExplorer create forms. Set the value to `create="false"` if you want to prevent this field from being available on DocExplorer create forms.
- **detail**. This attribute determines if the element is visible in DocExplorer *detail* forms. The value of this attribute must be either true or false. Set the value to `detail="true"` if you want this field to be available on DocExplorer detail forms. Set the value to `detail="false"` if you want to prevent this field from being available on DocExplorer detail forms.
- **label**. This attribute determines what name the element has in DocExplorer Available Field list. The value of this attribute can be any text string. Typically, you will want to set this value to a user-friendly name describing the content of the field.
- **list**. This attribute determines if the element is visible in DocExplorer *list* forms. The value of this attribute must be either true or false. Set the value to `list="true"` if you want this field to be available on DocExplorer list forms. Set the value to `list="false"` if you want to prevent this field from being available on DocExplorer list forms.

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

Use in physical mapping

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

Required Attributes

- **name.** This attribute determines the XML tag in which the Archway Document Manager places query results. The value of this attribute must match an element defined in the logical mapping section.
- **field.** This attribute identifies the field in the back-end database that you want the schema to use for document queries. The value of this attribute must be the SQL name of the field you want to use for the data source.

Optional attributes

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

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

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

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

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

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

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

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

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

- **linktype.** This attribute defines how the Archway Document Manager performs document inserts and updates. The value of this attribute must be either `soft` or `hard`:
 - `soft`—The Archway Document Manager queries the back-end database using the locations listed in the `linktable` and `linkfield` attributes, and sets the `link` attribute to the value to the query result.
 - `hard`—The Archway Document Manager creates a new record in the back-end database at the location listed in the `linktable` and `linkfield` attributes. The Archway Document Manager retrieves the `linkkey` value for the new record and saves it in the field listed in the `link` attribute.

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

<collection>

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

Use in logical mapping

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

Required attributes

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

Optional attributes

- **ACLcreate.** This attribute determines the default access control list for DocExplorer forms that use this subdocument. The value of this attribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see a **Create** button in DocExplorer forms that use this schema.
- **ACLdelete.** This attribute determines the default access control list for DocExplorer forms that use this subdocument. The value of this attribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will see a **Delete** button in DocExplorer forms that use this schema.
- **ACLupdate.** This attribute determines the default access control list for DocExplorer forms that use this subdocument. The value of this attribute must be a capability word. Users who meet or exceed the capability word listed in this attribute will be able to edit fields in DocExplorer detail forms that use this schema.
- **create.** This attribute determines if a subdocument using this element is visible in DocExplorer *create* forms. The value of this attribute must be either true or false. Set the value to `create="true"` if you want this subdocument to be available on DocExplorer create forms. Set the value to `create="false"` if you want to prevent this subdocument from being available on DocExplorer create forms.
- **detail.** This attribute determines if a subdocument using this element is visible in DocExplorer *detail* forms. The value of this attribute must be either true or false. Set the value to `detail="true"` if you want this subdocument to be available on DocExplorer detail forms. Set the value to `detail="false"` if you want to prevent this subdocument from being available on DocExplorer detail forms.

- **label.** This attribute determines what name the subdocument has in DocExplorer forms that use this schema. The value of this attribute can be any text string. Typically, you will want to set this value to a user-friendly name describing the content of the schema.
- **list.** This attribute determines if a subdocument using this element is visible in DocExplorer list forms. The value of this attribute must be either true or false. Set the value to `list="true"` if you want this subdocument to be available on DocExplorer list forms. Set the value to `search="false"` if you want to prevent this subdocument from being available on DocExplorer list forms.
- **search.** This attribute determines if a subdocument using this element is visible in DocExplorer *search* forms. The value of this attribute must be either true or false. Set the value to `search="true"` if you want this subdocument to be available on DocExplorer search forms. Set the value to `search="false"` if you want to prevent this subdocument from being available on DocExplorer search forms.

Use in physical mapping

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

Required attributes

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

Optional attributes

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

Documents

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

- **Schema logical definitions.** The schema logical definitions determine what XML elements make up the generated document.
- **The return values of database queries.** The Archway Document Manager uses the schema physical mappings to create database queries. The return values of these queries determine the content of the elements and attributes of the generated document.

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

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

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

- **Script Input.** This tab displays the document submitted to the current form from the output of a previous form. For example, a list form displays the output of a prior search form. This document is passed to the form onload script as an input parameter.
- **Script Output.** This tab displays the document generated by the output of the current form's onload script. Typically, each onload script invokes a schema that queries the back-end database for relevant information. For example, a request form will invoke a database query through the request schema.
- **PreXSL.** This tab displays the document after the Archway servlet has processed the document and prepared it to be rendered by the client-side browser.

Subdocuments

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

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

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

The following sections examples of each method.

Creating subdocuments with the <Document> element

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

Schema

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

```

<documents name="base">
  <document name="incident" label="Call"...>
    <attribute name="Id" type="id" label="Ticket Number".../>
    <attribute name="ProblemId" type="string" label="Problem Id".../>
    <attribute name="AssetTag" type="string" label="Asset Tag"/>
    ...
  <document name="Contact" docname="ticketcontact".../>
  ...
</document>
</documents>

<documents name="sc">
  <document name="incident" table="incidents"...>
    <attribute name="Id" field="incident.id"/>
    <attribute name="ProblemId" field="problem.id"/>
    <attribute name="AssetTag" field="affected.item"/>
    ...
  <document name="Contact" field="contact.name" table="contacts"
    joinfield="contact.name" joinvalue="ContactName"/>
  ...
</document>
</documents>

```

Logical mapping for
subdocument – Contact ———

Physical mapping for
subdocument – Contact ———

XML Output

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

```

Elements from schema      <incident>
mapping – Id, AssetTag    <Id>CALL10013</Id>
                          <AssetTag>TRAIN pc 100</AssetTag>
                          ...
Subdocument – Contact     <Contact>
                          <Id>Hartke</Id>
                          <FirstName>Richard</FirstName>
                          <LastName>Hartke</LastName>
                          <Email>Richard.Hartke@peregrine.com</Email>
                          <Phone>619-481-5000</Phone>
                          <Location/>
                          <LocationId/>
                          <UserAssets _countFound="0"/>
                          </Contact>
                          ...
                          </incident>

```

Creating subdocuments with the <Collection> element

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

Schema

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

```

<documents name="base">
  <document name="incident" label="Call"...>
    <attribute name="Id" type="id" label="Ticket Number".../>
    <attribute name="ProblemId" type="string" label="Problem Id".../>
    <attribute name="AssetTag" type="string" label="Asset Tag"/>
    ...
    <collection name="RelatedIncidents" detail="true"
      label="Related Incidents" ACLDelete="oaa.forbidden">
      <document name="relatedproblem" detail="true"
        subtypeprop="inherit" />
    </collection>
    ...
  </document>
</documents>

<documents name="sc">
  <document name="incident" table="incidents"...>
    <attribute name="Id" field="incident.id"/>
    <attribute name="ProblemId" field="problem.id"/>
    <attribute name="AssetTag" field="affected.item"/>
    ...
    <collection name="RelatedIncidents" >
      <document name="relatedproblem" table="screlation"
        joinfield="source" joinvalue="id" />
    </collection>
    ...
  </document>
</documents>

```

Logical mapping for subdocuments –
relatedproblem —————

Physical mapping for subdocuments –
relatedproblem —————

XML Output

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

```

Elements from schema mapping - Id, AssetTag      <incident>
                                                    <Id>CALL10013</Id>
                                                    <AssetTag>TRAIN pc 100</AssetTag>
Subdocument collection
-RelatedIncidents                               <RelatedIncidents _count="-1" _countFound="2" _more="0" _start="0">
                                                    <relatedproblem>
                                                    <Source>CALL10013</Source>
                                                    ...
                                                    <Id>CALL10013/IM10003</Id>
Subdocuments - relatedproblem                   <rincident>
                                                    <Id>IM10003</Id>
                                                    ...
                                                    </relatedproblem>
                                                    <relatedproblem>
                                                    <Source>CALL10013</Source>
                                                    ...
                                                    <Id>CALL10014/IM10004</Id>
                                                    <rincident>
                                                    <Id>IM10004</Id>
                                                    ...
                                                    </relatedproblem>
                                                    <relatedproblem>
                                                    </RelatedIncidents>
</incident>

```


9 Using Get-Services Tailoring

CHAPTER

This chapter describes methods and best practices for tailoring your Get-Services project.

This chapter covers the following topics:

- *Adding data validation to fields* on page 176
- *Assigning default values to fields with a custom script function* on page 185
- *Changing the strings displayed by priority, severity, or status fields* on page 192
- *Removing display values for priority, severity, or status* on page 195

Adding data validation to fields

You can have Get-Services validate field values in one of two ways:

- Make an input field required. Users will not be able to submit a form until they have entered all required fields.
- Add a custom validation script or function. If you want to check the validity of the data users submit, you must create a validation script or function.

Making a field required

Personalization forms allows you to mark fields as required, forcing users to fill in a value for that field in order to proceed to the following page. You can use required fields to assure that users complete all the fields that ServiceCenter requires when creating an incident or problem ticket. For example, you can make every Call ticket (in the Service Management module) require a user to attach an asset to it.

To make a field required:

- 1 Login to Get-Services with a user account that has `getit.personalization.admin` rights.
The user must have advanced personalization rights to save changes as default.
- 2 Navigate to the form you want to personalize, and then click the personalization wrench icon.
The Personalize Document Detail window opens.
- 3 From the Current Configuration window, double-click the field or subdocument that you want to require.
The field or subdocument properties window opens.
- 4 Select one of the following options:
 - *Subdocument*. For a subdocument, select the **Required** option under the Explorer Options section.
 - *Field*. For a field, toggle the **Required** option to **Yes**.
- 5 Click **Set as Default** to save your changes as the default view for all users.
All users who can see this form now see the required field or lookup.

Adding data validation with a custom script function

If you need to validate the actual data submitted on a Get-Services form, you can create a custom script function to check for certain conditions.

To add field validation with a custom script function:

- Step 1** Identify the schema that the form uses. See *Identifying the schema a form uses* on page 177.
- Step 2** Locate the schema file. See *Locating the schema file* on page 178
- Step 3** Identify the loadscript, if any, that the schema runs. See *Identifying the schema loadscript* on page 178.
- Step 4** Create a schema extension that calls a custom onload script file. See *Creating a schema extension to call a custom loadscript* on page 179.
- Step 5** Create a custom onload script file. See *Creating a custom onload script file* on page 181.

Identifying the schema a form uses

You can identify the schema used by a particular form directly from the Get-Services interface. Typically each form uses only one schema, but in some cases a form will use a subdocument that references another schema. The following procedures will help you determine what schema a particular form uses.

To identify the schema used by a particular form:

- 1** Enable Display form information from the **Administration > Settings** page. The Form information button displays in the banner bar of the Get-Services interface.
- 2** Browse to the form that you want to tailor.
- 3** Click the Form information button.
The form information window opens.
- 4** Search for one of the following entries on the Script Input tab:
 - `_docExplorerContext`. The last value listed after a slash in this element is the schema name. For example:
`<_docExplorerContext>incident/ticketcontact</_docExplorerContext>`
 uses the `ticketcontact.xml` schema file.

Note: In this example, `ticketcontact.xml` is a subdocument of the primary schema document `incident.xml`. Only DocExplorers will use this *document/subdocument* format.

- `_ctxschema`. The value listed in this element is the schema name. For example:
`<_ctxschema>ticketcontact</_ctxschema>`
 uses the `ticketcontact.xml` schema file.
- `_docExplorerSubType`. The value listed in this element is the schema name. For example:
`<_docExplorerSubType>TicketCreate</_docExplorerSubType>`
 uses the `TicketCreate.xml` schema file.
- `document`. The value listed in this element is the schema name. For example:
`<document>savedRequest</document>`
 uses the `savedRequest.xml` schema file.

If the schema name you find contains an underscore character, for example, `problem_search`, then this schema extends another existing schema, and the `loadscript` value you need resides in another schema file.

To determine the parent schema name, open the extending schema, and search for the attribute `extends`. The value of this attribute is the name of the parent schema. For example, the `problem_search` schema has the value `extends="problem"` and therefore extends the `problem` schema.

Locating the schema file

After you have determined the name of the schema that your forms uses, you can find it using your operating system's file search function. The following guidelines are provided to help narrow down your search:

- All schemas files have a `.XML` extension
- All schemas files are stored in the `WEB-INF\apps` folder of your application server's deployment directory. For example:
`C:\Program Files\Peregrine\Common\Tomcat4\webapps\oaa`

Identifying the schema loadscript

The schema `loadscript` is always listed in the logical mapping of the schema `<document>` element.

To identify the schema loadscript:

- 1 Open the schema XML source file of the schema that you previously identified in any text editor.
- 2 Locate the logical mapping section contained within the <documents name="base"> element.
- 3 Locate the logical mapping <document> element.
- 4 Locate the loadscript attribute.

The name of the loadscript is listed in quotation marks for this attribute. For example:

```

Schema loadscript ———— <documents name="base">
                          <document  name="problem" label="Ticket"
                          ACLcreate="getit.service" ACLdelete="oaa.forbidden"
                          ACLupdate="getit.service"
                          loadscript="preload.problem.imPreload"
                          preexplorer="preexplorer.problem.imPreexplorer">
                          ...
                          </document>
                          </documents>

```

Creating a schema extension to call a custom loadscript

Creating a schema extension allows you to change the loadscript without changing the original source code for Get-Services.

To create a schema extension to call a custom load script:

- 1 Copy the schema XML source file for the schema that you previously identified. For example, problem.xml.
- 2 Create a new folder as follows:

Create an extensions folder in the same directory where you found the source schema. For example:

```
C:\Program Files\Peregrine\Common\Tomcat4\webapps\oaa\WEB-INF\
  apps\incidentmgt\Schemas\extensions
```

- 3 Paste a copy of the source schema file in each of the folder you created.
- 4 Open the schema extension file in the extension folder.
This file is for your schema extension logical mappings.
- 5 Delete all the derived definitions listed in the bottom half of the original schema.

The derived definitions section starts after the first `</documents>` element and usually has a comment section describing what back-end databases and versions the derivations apply to.

Important: You must keep the `<?xml>` and `<schema>` elements.

- 6 Locate the logical mapping `<document>` element.

This element will be directly beneath the `<documents name="base">` element.

- 7 Record the value of the existing onload script attribute. For example:

```
Existing schema
loadscript -----<?xml version="1.0"?>
                  <schema>
                    <documents name="base">
                      <document name="problem" label="Ticket"
                        ACLcreate="getit.service" ACLdelete="oaa.forbidden"
                        ACLupdate="getit.service"
                        loadscript="preload.problem.imPreload"
                        preexplorer="preexplorer.problem.imPreexplorer">
                      </document>
                    </documents>
                  </schema>
```

If you were changing the problem schema, you would record the loadscript name `preload.problem.imPreload`.

You will need to know this loadscript name when you create your custom loadscript in the next section.

- 8 Edit the existing `<document>` element and change the value of loadscript attribute to the name of your custom loadscript. For example:

```
New schema
loadscript -----<?xml version="1.0"?>
                  <schema>
                    <documents name="base">
                      <document name="problem" label="Ticket"
                        ACLcreate="getit.service" ACLdelete="oaa.forbidden"
                        ACLupdate="getit.service"
                        loadscript="preload.custom.imPreload"
                        preexplorer="preexplorer.problem.imPreexplorer">
                      </document>
                    </documents>
                  </schema>
```

Steps for creating the custom onload script file are in the next section.

- 9 Save your changes to the schema extension file.

If all you are doing is changing the onload script called by your schema, then you do not need to create a schema extension for the physical mappings. See *Schema extensions* on page 132 for more information about schema extensions.

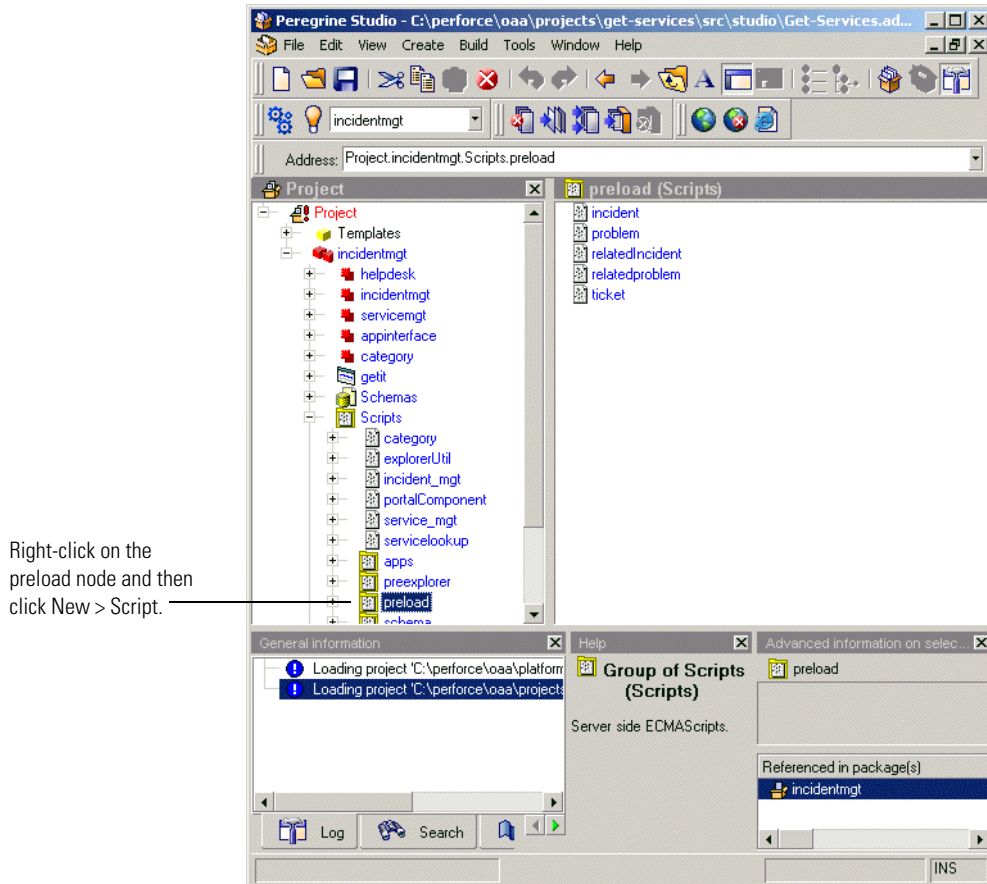
Creating a custom onload script file

You can create your own custom onload script to check incoming and outgoing data for certain conditions. You must create a custom script in order to not overwrite any of the scripts delivered with Get-Services.

To create a custom onload script:

- 1 Open the Get-Services project file in Peregrine Studio.
- 2 Expand the group of modules node for the portion of Get-Services you want to change.
 - **incidentmgt.** The group of modules for incident management. This includes the **Service Desk** tab in the Get-Services interface.
 - **changemgt.** The group of modules for change management. This includes the **Change Management** tab in the Get-Services interface.
- 3 Expand the **Scripts** node.

4 Right-click the preload node, and then click New > Script.



- 5 Give your script a unique name. For example, custom.
- 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 the original schema onload script. For example:

```
import the original
onload script _____ import preload.problem;
```

The script `preload.problem` is the name of the original onload script file called by the `problem` schema.

- 8 Right-click your new script node, and then click New > Function.

- 9 Rename your function to the desired custom onload function name. For example, `imPreload`.
- 10 Use the Peregrine Studio text editor to add a ECMAScript function to call your data validation functions and then call the original schema onload function for this form. For example:

```

function imPreload(msg)
{
  this.verifyFields(msg);
  if ( msg.testCondition("error", false) )
  {
    return msg;
  }
  msg = preload.problem.imPreload( msg );
  return msg;
}

```

Add a call to your data verification function —

Add a call to the original schema onload script —

- 11 Right-click your new script node, and then click **New > Function**.
- 12 Rename your function to the desired custom onload function name. For example, `verifyFields`.
- 13 Use the Peregrine Studio text editor to add a ECMAScript function to check the data entered and submitted by this form. For example:

```

function verifyFields(msg)
{
  var action = msg.get("_docExplorerAction");
  var formName = msg.get("_formname");

  if(action == "create" && formName == "new" )
  {
    var priority = msg.get("Priority");
    if (priority == 1)
    {
      msg.setCondition("error");
      user.addMessage("Please call the IT Department at 555-1212 for Priority 1 Tickets");
    }
    return msg;
  }
}

```

Check if field Priority is set to value 1 —

Return error message if field Priority has value of 1 —

- 14 Save and build your Get-Services project.

Your new data validation script displays the next time that you go to your the form. For example:

The screenshot shows the 'Create New Ticket' form in the Peregrine Portal. The form is titled 'Create New Ticket' and includes a navigation menu with 'Home', 'Change Management', 'Administration', and 'Service Desk'. The 'Service Desk' menu is expanded, showing 'Create Tickets', 'Ticket Status', and 'Ticket History'. The 'Incident Management' menu is also expanded, showing 'Incidents Assigned to Me', 'Unassigned Incidents', 'Search for Incidents', and 'Create Incidents'. The main form area contains a message: 'Please call the IT Department at 555-1212 for Priority 1 Tickets. Please fill in the requested information and press the submit button.' Below this is the 'Ticket Details' section with a 'Priority' dropdown set to 'Priority 1' and a 'Description' text area containing 'My PC is not booting up.'. There are also sections for 'File Attachments', 'Contact' (with 'Contact' set to 'roxy'), 'Asset Assigned to Ticket' (with 'Asset', 'Asset Type', 'Vendor Name', and 'Model' fields), and 'Submit Changes' and 'Go Back' buttons. Annotations on the left side point to the 'Error message displayed' and 'Priority 1 selected'.

Assigning default values to fields with a custom script function

You can easily assign default values to fields using a custom preprocess script. A preprocess script runs before the Archway Document Manager queries the back-end database. Since the Get-Services incident and problem schemas already have existing preprocess scripts, you must create a schema extension to call a custom script.

To assign default values to fields with a custom script function:

- Step 1** Identify the schema that the form uses. See *Identifying the schema a form uses* on page 185.
- Step 2** Locate the schema file. See *Locating the schema file* on page 186
- Step 3** Identify the preprocess script, if any, that the schema runs. See *Identifying the schema preprocess script* on page 187.
- Step 4** Create a schema extension that calls a custom preprocess script file. See *Creating a schema extension to call a custom preprocess script* on page 187.
- Step 5** Create a custom preprocess script file. See *Creating a custom onload script file* on page 181.

Identifying the schema a form uses

You can identify the schema used by a particular form directly from the Get-Services interface. Typically each form uses only one schema, but in some cases a form will use a subdocument that references another schema. The following procedures will help you determine what schema a particular form uses.

To identify the schema used by a particular form:

- 1** Enable Display form information from the **Administration > Settings** page. The Form information button displays in the banner bar of the Get-Services interface.
- 2** Browse to the form that you want to tailor.
- 3** Click the Form information button.
The form information window opens.
- 4** Search for the following text on the Script Input tab:

- `_docExplorerContext`. The last value listed after a slash in this element is the schema name. For example:
`<_docExplorerContext>incident/ticketcontact</_docExplorerContext>`
 uses the `ticketcontact.xml` schema file.

Note: In this example, `ticketcontact.xml` is a subdocument of the primary schema document `incident.xml`. Only DocExplorers will use this *document/subdocument* format.

- `_ctxschema`. The value listed in this element is the schema name. For example:
`<_ctxschema>ticketcontact</_ctxschema>`
 uses the `ticketcontact.xml` schema file.
- `_docExplorerSubType`. The value listed in this element is the schema name. For example:
`<_docExplorerSubType>TicketCreate</_docExplorerSubType>`
 uses the `TicketCreate.xml` schema file.
- `document`. The value listed in this element is the schema name. For example:
`<document>savedRequest</document>`
 uses the `savedRequest.xml` schema file.

If the schema name you find contains an underscore character, for example, `problem_search`, then this schema extends another existing schema, and the `loadscript` value you need resides in another schema file.

To determine the parent schema name, open the extending schema, and search for the attribute `extends`. The value of this attribute is the name of the parent schema. For example, the `problem_search` schema has the value `extends="problem"` and therefore extends the `problem` schema.

Locating the schema file

After you have determined the name of the schema that your forms uses, you can find it using your operating system's file search function. The following guidelines are provided to help narrow down your search:

- All schemas files have a `.XML` extension
- All schemas files are stored in the `WEB-INF\apps` folder of your application server's deployment directory. For example:
`C:\Program Files\Peregrine\Common\Tomcat4\webapps\oaa`

Identifying the schema preprocess script

The schema preprocess script is always listed in the physical mapping of the schema <document> element.

To identify the schema loadscript:

- 1 Open the schema XML source file of the schema that you previously identified in any text editor.
- 2 Locate the physical mapping section contained within the <documents name="adapter_name"> element.
- 3 Locate the physical mapping <document> element.
- 4 Locate the preprocess attribute.

The name of the preprocess script is listed in quotation marks for this attribute. For example:

```

Schema preprocess
script ————— <documents name="sc" ...>
                    <document name="incident" table="incidents" insert="esmin"
                    update="esmin" postprocess="schema.incident.scPostprocess"
                    preprocess="schema.incident.scPreprocess">
                    ...
                    </document>
                    </documents>

```

Creating a schema extension to call a custom preprocess script

Creating a schema extension allows you to change the preprocess script without changing the original source code for Get-Services.

To create a schema extension to call a custom preprocess script:

- 1 Copy the schema XML source file for the schema that you previously identified. For example, incident.xml.
- 2 Create a new folder as follows:

Create an extensions folder in the same directory where you found the source schema. For example:

```
C:\Program Files\Peregrine\Common\Tomcat4\webapps\oaa\WEB-INF\apps\incidentmgt\Schemas\extensions
```
- 3 Paste a copy of the source schema file in each of the folder you created.
- 4 Open the schema extension file in the extension folder.

This file is for your schema extension logical mappings.
- 5 Delete all the logical definitions listed in the top half of the original schema.

The logical definitions section starts after the first <documents> element.

Important: You must keep the <?xml> and <schema> elements.

6 Locate the physical mapping <document> element.

This element will be directly beneath the <documents name="adapter_name"> element.

7 Record the value of the existing preprocess script attribute. For example:

```

Existing schema
preprocess script ————<?xml version="1.0"?>
                        <schema>
                        <documents name="sc" ...>
                          <document name="incident" table="incidents" insert="esmin"
                          update="esmin"
                          postprocess="schema.incident.scPostprocess"
                          preprocess="schema.incident.scPreprocess">
                            ...
                          </document>
                        </documents>
                        <schema>

```

If you were changing the problem schema, you would record the loadscript name `schema.incident.scPreprocess`.

You will need to know this preprocess script name when you create your custom preprocess script in the next section.

8 Edit the existing <document> element and change the value of preprocess attribute to the name of your custom preprocess script. For example:

```

New schema
preprocess script ————<?xml version="1.0"?>
                        <schema>
                        <documents name="sc" ...>
                          <document name="incident" table="incidents" insert="esmin"
                          update="esmin"
                          postprocess="schema.incident.scPostprocess"
                          preprocess="schema.custom.scPreprocess">
                            ...
                          </document>
                        </documents>
                        <schema>

```

Steps for creating the custom onload script file are in the next section.

9 Save your changes to the schema extension file.

If all you are doing is changing the onload script called by your schema, then you do not need to create a schema extension for the physical mappings. See *Schema extensions* on page 132 for more information about schema extensions.

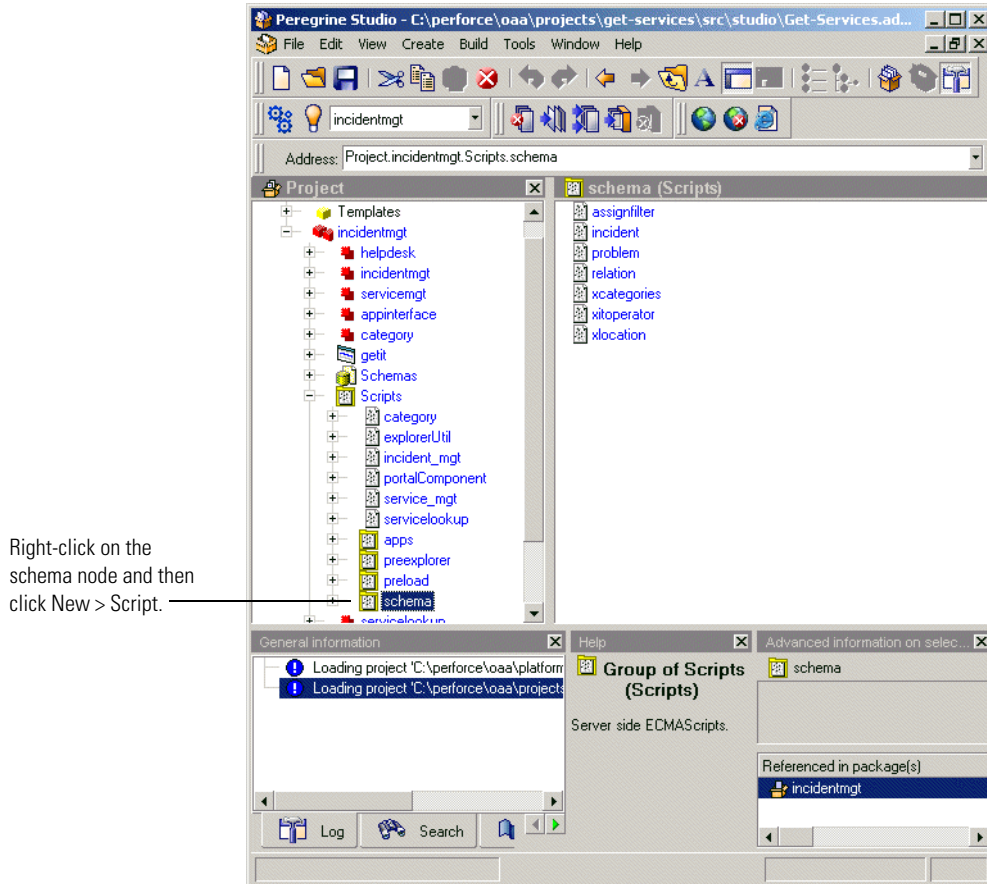
Creating a custom preprocess script file

You can create your own custom preprocess script to assign default values to fields in the XML document. You must create a custom script in order to not overwrite any of the scripts delivered with Get-Services.

To create a custom preprocess script:

- 1 Open the Get-Services project file in Peregrine Studio.
- 2 Expand the group of modules node for the portion of Get-Services you want to change.
 - **incidentmgt.** The group of modules for incident management. This includes the **Service Desk** tab in the Get-Services interface.
 - **changemgt.** The group of modules for change management. This includes the **Change Management** tab in the Get-Services interface.
- 3 Expand the **Scripts** node.

- 4 Right-click the schema node, and then click New > Script.



- 5 Give your script a unique name. For example, custom.
- 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 the original schema onload script. For example:

import the original
preprocess script _____ import schema.scPreprocess;

The script `schema.scPreprocess` is the name of the original preprocess script file called by the incident schema.

- 8 Right-click your new script node, and then click New > Function.

- 9 Rename your function to the desired custom onload function name. For example, `scPreprocess`.
- 10 Use the Peregrine Studio text editor to add a ECMAScript function to define any default values and then call the original schema onload function for this form. For example:

```

function scPreprocess(msg)
{
  assignDefaultValues(msg);
}

```

Create a function call –
 assignDefaultValues _____

```

msg = schema.incident.scPreProcess( msg );

```

Add a call to the _____
 original schema onload _____
 script

- 11 Right-click your new script node, and then click **New > Function**.
- 12 Rename your function to the desired custom onload function name. For example, `assignDefaultValues`.
- 13 Use the Peregrine Studio text editor to add a ECMAScript function to add default values submitted to the back-end database. For example:

```

function assignDefaultValues(msg);
{
  var strTemp = msg.get("Category");
  if (strTemp == "")
  {
    msg.set("Category", "business applications");
  }
}

```

Check for existing
 value of Category _____

Assign Category _____
 default value of
 "business applications" _____

- 14 Save and build your Get-Services project.

Changing the strings displayed by priority, severity, or status fields

Get-Services displays the **Priority**, **Severity**, and **Status** values defined in the project string resources. You can change the value Get-Services displays for each option by changing the value of the string resource.

To change the strings displayed by priority, severity, or status fields:

- 1 Open the Get-Services project file in Peregrine Studio.
- 2 Expand the group of modules node for the portion of Get-Services you want to change.
 - **incidentmgt**. The group of modules for incident management. This includes the **Service Desk** tab in the Get-Services interface.
 - **changemgt**. The group of modules for change management. This includes the **Change Management** tab in the Get-Services interface.
- 3 Expand the **StringResources** node.
- 4 Use the following tables to locate the strings resources that you want to change.

Call severity string resources.

Value	String Resource	Default String Value
1	callSeverityCritical	Critical
2	callSeverityMajor	Major
3	callSeverityMedium	Medium
4	callSeverityLow	Low
5	callSeverityVeryLow	Very Low

Call status string resources.

Value	String Resource	Default String Value
1	serviceCallStatusOpen	Open
2	serviceCallStatusIdle	Open - Idle
3	serviceCallStatusCallback	Open - Callback

Value	String Resource	Default String Value
4	serviceCallStatusLinked	Open - Linked
5	serviceCallStatusClosed	Closed

Incident priority string resources.

Value	String Resource	Default String Value
1	serviceProblemPriority1	Priority 1
2	serviceProblemPriority2	Priority 2
3	serviceProblemPriority3	Priority 3
4	serviceProblemPriority4	Priority 4

Incident severity fields.

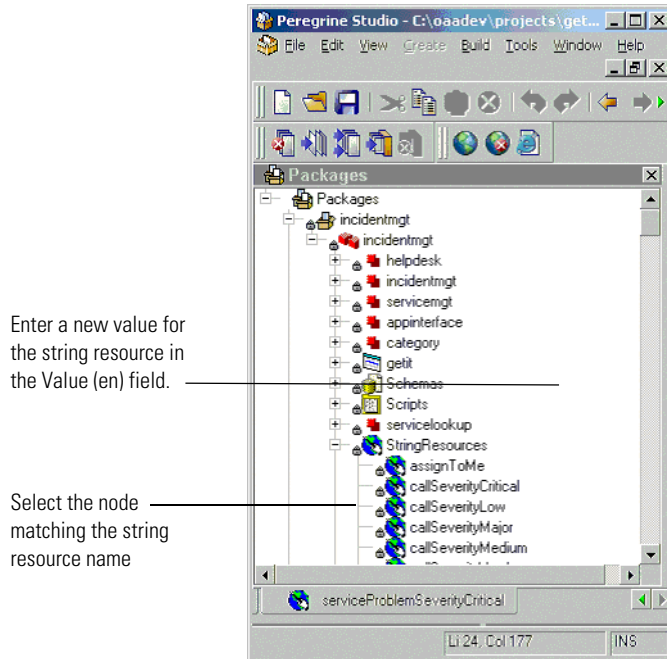
Value	String Resource	Default String Value
1	serviceProblemSeverityCritical	Critical
2	serviceProblemSeverityUrgent	Urgent
3	serviceProblemSeverityNormal	Normal
4	serviceProblemSeverityLow	Low
5	serviceProblemSeverityVeryLow	Very Low

Incident status string resources.

Value	String Resource	Default String Value
1	serviceProblemStatusOpen	Open
2	serviceProblemStatusWIP	Work in progress
3	serviceProblemStatusResolved	Resolved
4	serviceProblemStatusClosed	Closed

- 5 Select the node matching the string resource name that you want to change. For example, `callSeverityCritical`.

- 6 Type a new value in the Value (en) field. For example, System down.



- 7 Save your Get-Services project file.
- 8 Click **Build > Differential Build**.
Peregrine Studio generates the new field values.
- 9 Deploy the new build files to your deployment server.

Removing display values for priority, severity, or status

You can remove options from the **Priority**, **Severity**, and **Status** fields by directly editing the custom JSP files that generate these combo boxes.

To remove the Get-Services combo box display values:

- 1 Locate the custom JSP file used by the combo box component you want to change.

The JSP file called will be one of the following files:

File Name	Description
call_severitycombo.jsp	Displays the severity field in call tickets
call_statuscombo.jsp	Displays the status field in call tickets
problem_prioritycombo.jsp	Displays the priority field in incident tickets
problem_severitycombo.jsp	Displays the severity field in incident tickets
problem_statuscombo.jsp	Displays the status field in incident tickets

- 2 Create a back-up copy of any file you plan to edit.
- 3 Open the custom JSP file in any text editor.

By default, the custom JSP files are located in the application server deployment directory. For example:

- 4 Search for methods that include the string `cw.user.getIDSAWD`.

Each display value is listed twice in the JSP file. The first listing gets the string resource to display in the combo box. The second listing adds a value recognized by `ServiceCenter`. The first listing uses `if` and `else if` conditions. The second listing uses `vecValues` and `vecDisplays` methods.

For example, the call severity combo box has five entries:

```
String strSeverity = msgModel.get("Severity");
Each value has a condition mapped to a string resource ----- if( strSeverity.equals("1") )
{
    strSeverity = cw.user.getIDSADW(msgModel,
    "incidentmgt,callSeverityCritical");
}
else if( strSeverity.equals("2") )
{
Each string resource uses the call cw.user.getIDSADW ----- strSeverity = cw.user.getIDSADW(msgModel,
    "incidentmgt,callSeverityMajor");
}
else if( strSeverity.equals("3") )
{
    strSeverity = cw.user.getIDSADW(msgModel,
    "incidentmgt,callSeverityMedium");
}
else if( strSeverity.equals("4") )
{
    strSeverity = cw.user.getIDSADW(msgModel,
    "incidentmgt,callSeverityLow");
}
else if( strSeverity.equals("5") )
{
    strSeverity = cw.user.getIDSADW(msgModel,
    "incidentmgt,callSeverityVeryLow");
}
...
Each value has a vecValues and vecDisplays entry ----- vecValues.add( "1" );
    vecDisplay.add(cw.user.getIDSADW(msgModel,
    "incidentmgt,callSeverityCritical"));
vecValues.add( "2" );
    vecDisplay.add(cw.user.getIDSADW(msgModel,
    "incidentmgt,callSeverityMajor"));
vecValues.add( "3" );
    vecDisplay.add(cw.user.getIDSADW(msgModel,
    "incidentmgt,callSeverityMedium"));
vecValues.add( "4" );
    vecDisplay.add(cw.user.getIDSADW(msgModel,
    "incidentmgt,callSeverityLow"));
vecValues.add( "5" );
    vecDisplay.add(cw.user.getIDSADW(msgModel,
    "incidentmgt,callSeverityVeryLow"));
...

```

- Remove the condition reference for the option you want to remove. For example, to remove the severity 5 entry, remove the following code:

Remove the condition
and everything
between the curly
brackets {}

```

else if( strSeverity.equals("5") )
{
    strSeverity = cw.user.getIDSADW(msgModel,
    "incidentmgt,callSeverityVeryLow");
}

```

- Remove the `vecValues` and `vecDisplays` methods for the option you want to remove. For example, to remove the severity 5 entry, remove the following code:

Remove both the
`vecValues` and
`vecDisplays` method

```

vecValues.add( "5" );
vecDisplay.add(cw.user.getIDSADW(msgModel,
"incidentmgt,callSeverityVeryLow"));

```

- Save the custom JSP file.
- Restart your application server.

The new combo box values display in your Get-Services forms.

A

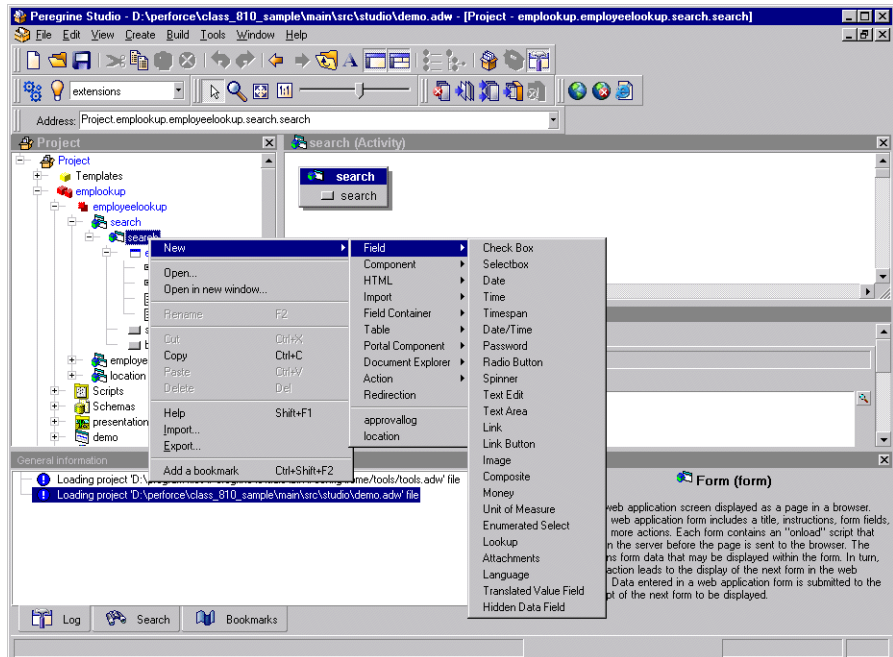
APPENDIX

Peregrine Studio Components

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

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

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



Project > New >
Directory Object—not supported.

Group of Modules > New >

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

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

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

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

Field >

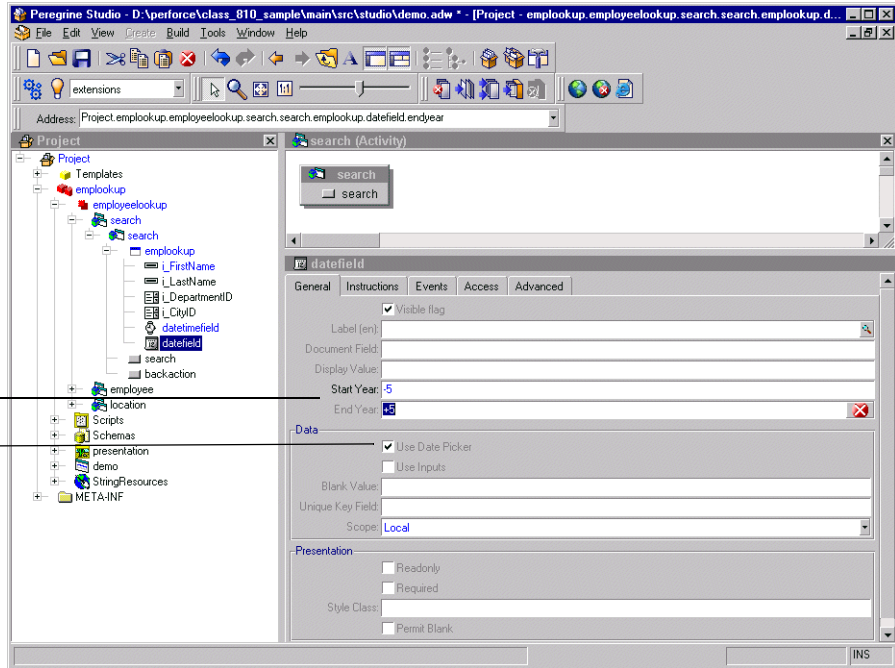
Check Box—Allows the user to toggle a value on or off.

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

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

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

Enable or disable the calendar widget.



Time—Allows the user to view or set a time value.

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

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

Password—Allows the user to enter a password.

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

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

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

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

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

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

Image—Displays an image.

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

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

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

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

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

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

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

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

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

Component >

Treelink—Displays a treelink component.

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

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

Workflow—Displays a workflow diagram.

OAA Workflow—Displays a workflow diagram.

Stack—Displays a stack component.

SVG—Displays an SVG component.

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

HTML >

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

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

Import >

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

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

Field Container >

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

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

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

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

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

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

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

Table >

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

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

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

Portal Component >

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

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

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

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

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

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

Document Explorer >

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

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

Detail—Displays a personalized view of a document detail.

Action >

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

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

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

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

Print—Prints the current Get-Services form.

Close—Use to close pop-up windows.

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

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

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

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

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

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

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

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

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

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

- **Group of Scripts**—Server-side ECMAScripts.
- **Group of Triggers**—A collection of triggers. Used by applications using BizDoc.
 - **Trigger**—Individual trigger for a document.
 - Message action**—Message action executed by the trigger.
 - Workflow action**—Workflow action executed by the trigger.
 - Script action**—Script action executed by the trigger.
 - Bizdoc Java action**—Java action executed by the trigger inside Bizdoc.
 - **Group of Triggers**—Collection of triggers.
 - Trigger**—Individual trigger for a document.
 - Group of Triggers**—Collection of triggers.
- **Group of Schemas**—Database schemas describing documents accessible by Get-Services. Schemas define the field table mapping between Get-Services and the back-end database.
 - **Raw Schema**—Description of a document's mapping on a real database.
 - **Schema**—not supported.
- **Group of Images**—Folder containing the image files to be used in your Web application.
 - **Image**—The image is loaded into the ImageData property as binary data. The file name property is used only the first time to load the image.
 - **Group of Images**—Folder containing image files.
 - Image**—The image is loaded into the ImageData property as binary data. The file name property is used only the first time to load the image.
 - Group of Images**—Folder containing image files.
- **Group of Presentation Files**—Folder containing files copied directly to the presentation folder for use within the Get-Services Web server.

- **Text Presentation File**—Any generic file in the Presentation folder that is needed by the Web server, for example, client-side JavaScript, static JSP files.
- **Binary Presentation File**—Binary file outputted in the presentation folder. Accessed by the Web server and used by the browser.
- **Group of Presentation Files**—Folder containing files copied directly to the presentation folder for use within the Get-Services Web server.
 - Text Presentation File**—Any generic file in the Presentation folder that is needed by the Web server, for example, client-side JavaScript, static JSP files.
 - Binary Presentation File**—Binary file outputted in the presentation folder. Accessed by the Web server and used by the browser.
 - Group of Presentation Files**—Folder containing files copied directly to the presentation folder for use within the Get-Services Web server.
- **Group of default DocExplorer screens**—Folder containing default content for DocExplorer Personalization screens.
 - **Reference of a file**—File object.
 - **Directory Object**—not supported.
- **Group of Portal Components**—Components that appear in the portal components menu and can be added to the home page by the user.
 - **Portal Component**
 - (**contents**)—The content of the portal component that is displayed.
 - (**configure**)—Allows configuration of a portal component.
- **Group of Files**—A temporary container of miscellaneous files used by a Web application. For example, string files and scriptpoller.ini files are stored here.
 - **String file**—Temporary representation of a string file.
 - **Ini file**—Temporary representation of a scriptpoller.ini file.
- **Group of Strings**—List of multilingual strings.
 - **Multilingual String**—The name of the StringResource is the ID of the string.
- **Group of Roles**—not supported.

Group of Style Sheets > New >

- Style Sheet—Not supported.

Group of Roles—not supported.**Group of Files > New >**

- String file—Temporary representation of a string file.
- Ini file—Temporary representation of a scriptpoller.ini file.

Group of Strings > New >

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

Entities (collection of business objects) > New >

- Entity—Used by applications using BizDoc.

Interfaces > New >

- Interface—Not supported.

System enumerations > New >

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

Templates > New

- Schema >Not supported.
- Field Container
 - Component Template
- Directory Object—not supported.
- Group of Methods—Includes a list of methods. You can create new methods under this element.
 - Method—Java Method. The name is not significant. You can add a comment to the method.
- Method—Java Method. The name is not significant. You can add a comment to the method.
- Message action—Message Action executed by the trigger.

- **Workflow action**—Workflow action executed by the trigger.
- **Bizdoc Java action**—Java action executed by the trigger inside Bizdoc.
- **Script action**—Script action executed by the trigger.
- **Trigger**—Used by applications using BizDoc.
- **Group of Images**—Allows you to create a group of images.
- **Attribute**—Ejb attribute. Used by BizDoc.
- **Reference**—Ejb reference. Used by BizDoc.
- **Contain**—Contain an object as an embedded member.
- **Computed**—Computed property.
- **Structure**—Ejb structure. Used by BizDoc.
- **Collection**—Ejb collection. Used by BizDoc.
- **Methods**—Ejb method. Used by BizDoc.
- **Entity**—Ejb entity. Used by BizDoc.

B Troubleshooting and FAQs

CHAPTER

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

This chapter covers the following topics:

- *Get-Resources Environment* on page 212
- *Peregrine Studio* on page 213
- *Scripting Errors* on page 216
- *Tailoring Errors* on page 218

Get-Resources Environment

This section describes warnings or errors that can be generated while running a Get-Services in your system environment.

Out of memory error

Problem

Your application server has run out of memory resources.

Solution

Get-Services run best on a system with a minimum of 512 MB of RAM. If you cannot add more physical memory to your machine, you can increase the virtual memory space used on your Windows system. Adding virtual memory will require more hard disk space and may degrade system performance as cached information is saved to and retrieved from the hard disk. Refer to your Windows help for information on setting or changing virtual memory.

Cannot start Java – JRE must be installed

Problem

Peregrine Studio produces an error message when you attempt to create a package or build a project.

```
Cannot start Java ('jvm.dll' not found). The JRE (Java Runtime Environment) must be installed ...
```

Solution

Install a dedicated copy of the Java 2 SDK for Peregrine Studio to use. You can install the Java 2 SDK from the Get-Services installation CD.

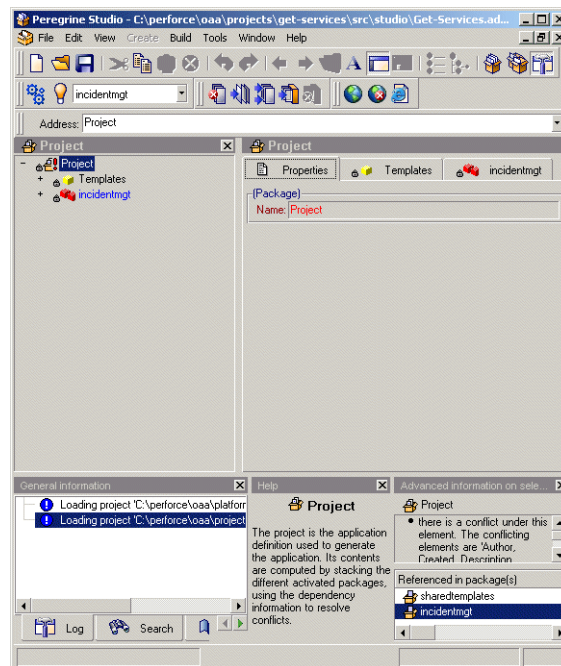
Peregrine Studio

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

Cannot edit — components are displayed with grey background

Problem

Peregrine Studio displays some or all of your project components with a grey background, and you cannot make or save changes to the project components.



Solution

Peregrine Studio uses the grey background to indicate that an item is write protected. The most common reasons that Peregrine Studio components are write protected are:

- A write-protected package is selected in the package selector.
- The project (.adw) file is set to read-only.

Packages delivered by Peregrine are write-protected. You must save all of your changes and additions to a user-created package extensions. If the package selection box displays one of the Peregrine Studio default packages, then your project will be write protected until you create and activate a new package extension in which to save your changes.

Red exclamation point (conflict icon) displayed next to nodes


Problem

Peregrine Studio displays a conflict icon next to one or more of your project components, and you cannot build the project. The conflict could be the result of multiple packages attempting to change or modify the same component, or the conflict could be the result of improperly defined package dependencies.

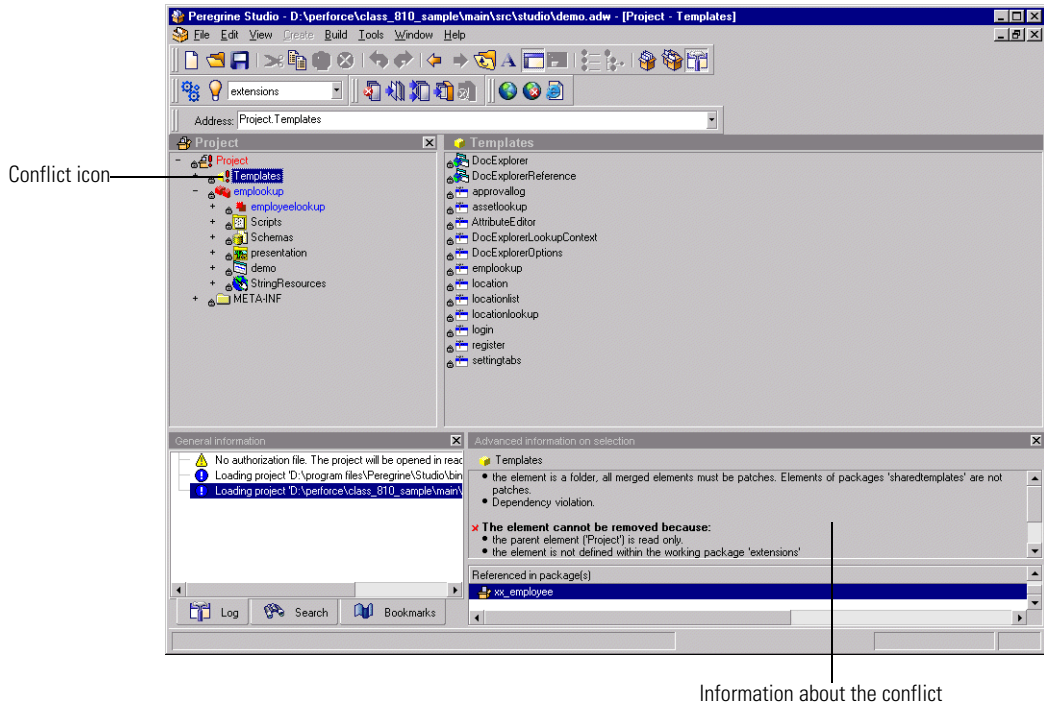
Solution

To resolve the conflict you should first view more information about the nodes displaying the conflict icon.

To view information about a conflict:

- 1 Select a node with an exclamation point  icon displayed next to the name from the Project Explorer view.
- 2 Click **View > Advanced Information**. Studio displays a new information window at the bottom of the interface. This window displays information on the conflict.

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



Resource conflicts

Resource conflicts occur when two or more project components describe the same thing. To resolve a resource conflict, delete or reconfigure one of the project components that is creating the conflict. If the conflicting components are part of separate package extensions, you can choose to deactivate one of the package extensions to resolve the conflict.

Dependency conflicts

Dependency conflicts occur when a package extension attempts to modify a package that is not listed as a dependent package. To resolve the conflict you can choose one of two solutions:

- Add the package you want to modify as a package dependency of the conflicting package extension.
- Move the changes in the conflicting package extension to another package extension that already has the proper package dependencies.

Scripting Errors

Information about scripting errors is displayed as text at the top of the main frame and in the `archway.log` file.

Unable to find script file

Problem

The following error message is displayed when you select a form:

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

This message will also appear in the `archway.log` file.

Solution

This error message is usually the result of a script file trying to call an undefined adapter. This is a common problem if you import a Web application into a project that contains a new adapter. Review your script file and determine what adapters it calls. If the `<name>` value is the name of a new adapter defined in the script file, then define the new adapter in the Admin Settings module, stop and restart your application server, and then restart the Archway server (using the Admin Control Panel) to correct the problem.

This error message can also appear if a form is calling an invalid script file name. Verify in Studio that the form is calling a valid script file name. If you copied a script from another form or Web application you may have renamed the script incorrectly.

If you have verified that the script file exists and uses the proper adapter, then stop and restart your application server. This will refresh the adapter settings.

Script produces an ECMAScript error

Problem

An ECMAScript Error is displayed with the script name, source code, and line number of the error when a form is displayed.

Solution

Open Peregrine Studio, review the error-producing script for typos, and verify that it uses the correct function and schema names. For example, you might have a function where *msg* is incorrectly listed as *nsg*. Correct any errors and rebuild the project.

Note: ECMAScript is case sensitive and will return an error message if the case does not match the object called.

Tip: If you have enabled the HTTP listener in Peregrine Studio, you can click on the underlined script name listed at the top of the error message to go directly to the script and line number of the error. Peregrine Studio must be open for the hyperlink to work.

ECMAScript error: undefined value or property

Problem

The following error is displayed when you select a form:

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

This error will also be displayed in the `archway.log` file.

Solution

Verify that the form calls the proper script name in the server onload script attribute. Also check that the script name contains no typos and that it is listed with the proper case. If the script name listed in the form is correct, there is a possibility that there is a script name conflict. Each script in your project needs a unique name. Try renaming your script to a new name, updating the server onload script attribute, and rebuilding your project. If renaming the script fixes the problem then you had a script name conflict.

Tailoring Errors

The following sections describe some of the common errors associated with tailoring Get-Services. Refer to the sections below for solutions to common tailoring problems.

Wrong start form is displayed for activity

Problem

You want Get-Services to display a particular form when you select an activity, but the wrong form is displayed. You may have also re-ordered the form listing in the Project Explorer tree, but the proper form still is not displayed.

Solution

You need to define the Start Page attribute of the activity. This attribute determines what form is first displayed when the activity is selected. By default, the Start Page is blank.

To set the Start Page of an activity:

- 1 Open Peregrine Studio and select the activity you want to change.
Tip: To select the activity properties, select the activity node, double-click any form in the flowchart view displayed, and then click the Control tab. The activities properties will be displayed to the right of the control flowchart.
- 2 In the properties of the activity, use the selectbox of the Start Page attribute to choose a starting form.
- 3 Save and rebuild your project file.

Script output not appearing in form component

Problem

Data is not displayed in your Get-Services 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-Services data encoded as in the XML document displayed on the Script Output page. If you do not see an XML document, then your script has an error.

If you can see data displayed in the Script Output tab, then the problem is how you have mapped the form components to the XML fields. View the form component properties from Peregrine Studio, and verify that the Document Field attribute of the form component maps to an XML tag displayed in the Script Output tab.

Too few parameters error

Problem

The following error message is displayed when you select a form:

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

The script output displays the following error:

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

These messages will also appear in the archway.log file.

Solution

There is an incorrect field mapping or typo in the schema used in this form. Review the schema(s) used by this form and verify that there are no typos. Also verify that all the attributes defined in the schema map to valid fields in the back-end database. The value in the field attribute must match the field name of the back-end database. This is particularly important for the ID attribute, which must map to a unique numerical value that identifies each record.

Get-Services always goes to redirection form

Problem

You have defined a redirection to another form in Get-Services and the source form always takes users to the redirection form regardless of the search conditions and results.

Solution

Validate that the Condition attribute of the redirection is not blank. The Condition value should match the value defined by the setCondition function of your form's ECMAScript. If the Condition attribute is left blank, the default action is to redirect to the target form regardless of the returned results.

Syntax error in FROM clause

Problem

The following error message is displayed when you select a form:

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

The script output displays the following error:

```
-3506 [...] Syntax error in FROM clause.
```

This error will also be displayed in the archway.log file.

Solution

The schema name you defined for the form is wrong. The schema name could be listed incorrectly in two places:

- The form's onload script may refer to the wrong schema name.
- The <document name=*value*> does not match the schema file name.

Index

A

- actions. See form components
- activity component 38
- Archway
 - scripts 102
- Archway Document Manager
 - and schemas 131
- authorization file
 - Peregrine Studio 18

B

- bookmarks, adding in Studio 28
- build options 42–43
 - build directory 43
 - character encoding 43
 - EJB user 43
 - exclude files 43
 - presentation folder 43
 - temporary directory 43

C

- cascading style sheets 36
- component template 74–75
- components
 - group of files component 39
 - group of modules component 38
 - group of schemas component 39
 - group of scripts component 39
 - hierarchy of 37
 - in Peregrine Studio 199
 - module component 38

- modules 41
 - package 41
 - relationships among 38–39
- conflicts
 - defined 47
 - resolving 47–48, 215
- creating
 - nested document lookups 93
 - package extensions 45
 - schemas 149

D

- data validation
 - methods of 176
 - tailoring tasks 24
 - using a custom script 177–184
- default values
 - assigning using custom script 185–191
 - tailoring tasks 24
- dependencies
 - setting for packages 46
- dependency conflicts. See Conflicts
- deployment directory 43
- development environment
 - requirements for 19
- DocExplorer Reference
 - adding 90
- DocExplorers
 - personalizing with 89
 - tailoring tasks 23
- Document field

format of names 65
document schema definitions. See schemas

E

ECMAScript 101

errors

- sysntax error in FROM clause 220
- too few parameters 219
- Unable to find script file 216
- undefined value or property 217

F

field labels, changing 60

fields

- making required 176

fields. See form components

fieldsection component 75

form component 38

form components

- action 38, 83–84
- changing schemas 63
- component template 74
- date picker 202
- described 38
- field form components 60
- fields 38
- fieldsection 75–76
- hidden data field 79
- hiding 61
- labels 60
- lookups 38
- making read-only 62
- names in 65–67
- redirection 80
- selectbox 77–79
- simple table 81
- table link 82
- tables 38
- tailoring 56–84
- tailoring tasks 23
- text columns 82–83
- text edit 76–77

forms

- changing instructions 58
- changing onload scripts 59

- changing titles 57

- server-side 102–103

- tailoring tasks 23

framesets

- displaying forms in 67

G

group of scripts component 39

H

HTTP Listener 30

HTTP listener

- enabling in Peregrine Studio 30

I

installation

- tailoring kit 14

instructions, changing in forms 58

interface components. See Form components 38

ISO character encoding. See character encoding

J

JavaDocs 128

JavaScript 101

L

lookup fields 92

- creating 92

- nested document lookups 93

lookups. See form components

M

messages, scripts 113

N

nodes 29, 214

- group of schemas node 149

O

onload scripts

- changing in forms 59

- defined 59

P

package extensions 45–47

- packages
 - activating 45
 - deactivating 46
 - defined 44
 - dependencies 46
 - Peregrine Studio
 - authorization file 18
 - Personalization
 - adding existing fields 97
 - configuring fields 98
 - interface, described 95
 - lookup fields 92
 - removing existing fields 97
 - requirements 86
 - settings 87
 - tailoring tasks 22
 - Portal components
 - Business View Authoring 88
 - creating 72
 - making schemas visible to 88
 - presentation files 36
 - priority values
 - changing 192
 - removing 195
 - Project Explorer 29
 - projects
 - See also Web applications
 - components of 36
 - conflicts within 48
 - files within 40
- R**
- resource conflicts. See conflicts
 - Rhino JavaScript Debugger 109–110
- S**
- schema elements
 - 166
 - schema template example 155
 - schemas
 - adding logical and physical mappings 149
 - Archway Document Manager and 131
 - changing in form components 63
 - creating 149
 - creating your own 148
 - defined 130
 - document fields 64
 - elements 156–173
 - extension folders 135
 - extensions 22, 132–147
 - identifying schema used 133
 - locating 134
 - sample 155
 - tailoring tasks 22, 24
 - testing from a URL 111–112
 - uses for extensions 136
 - using with DocExplorers 90
 - scripts
 - adding to Peregrine Studio project 107
 - client-side 100
 - creating XML message objects 113
 - displaying variables in form components 69
 - ECMAScript 101
 - editing 105
 - format of variables 69
 - JavaScript 101
 - list of references 128
 - object oriented usage 116
 - onload scripts 102–103
 - prototype property 116
 - roles of 102
 - samples 121–127
 - server scripts 101
 - server-side 100
 - tailoring tasks 23
 - testing from a URL 110–111
 - uses for 100
 - ServiceCenter 86
 - severity values
 - changing 192
 - removing 195
 - source files
 - opening in Peregrine Studio 18
 - status values
 - changing 192
 - removing 195
 - string files
 - translating 51, 52

T

tables. See form components

tailoring

form components 56–84

tailoring kit

installation 14

tailoring tasks

requiring Peregrine Studio 23

that can be done outside of Peregrine Studio
22

templates component 37

testing environment

requirements for 19

titles, changing in forms 57

translating

tailored modules 49

troubleshooting

cannot start Java 212

conflicts 214

JRE must be installed 212

Read-only components 213

redirections 220

script error 217

script error Unable to find script file 216

script error undefined value or property 217

sysntax error in FROM clause 220

too few parameters 219

virtual memory error 212

viewing changes 32

X

XML

components. See components

creating message objects from scripts 113

example of Document field names 66

example of script variable name 69

viewing source code 32

U

UNIX

deploying tailoring changes to 49

URL

querying scripts and schemas from 110

user rights, Personalization 87

V

variables

referring to XML attributes 71

visible flag

hiding form components 61

W

Web applications

described 10

