
Get It!

Tailoring Guide
Release 1.3
September 2000

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


S Y S T E M S[®]
The Infrastructure Management Company™

© 2000 by Peregrine Systems, Inc. 3611 Valley Centre Drive, San Diego, CA 92130 U.S.A.
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 is a registered trademark of Peregrine Systems, Inc.

Microsoft, Windows, and Windows NT are either trademarks or registered trademarks of Microsoft Corporation in the United States and/or other countries.

The Motive logo, Motive Communications, Motive Duet, Motive Duet Server, Motive Solo, Motive Solo Server, Motive Support Tuner, Motive Support Portal, Motive Studio, Motive Support Desktop, Motive ActiveLink, Motive ActiveSense, and Motive Integration Server are trademarks or registered trademarks of Motive Communications, Inc.

Java and JavaScript are trademarks of Sun Microsystems, Inc. Solaris is a registered trademark in the United States and other countries licensed exclusively through Sun Microsystems. The code examples provided herein are for illustrative purposes only and are not intended for actual customer use without additional customization and proper testing prior to deployment.

This document and the related software described in this manual is 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.

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.

This edition applies to version 1.3 of the licensed program.

Contents



Contents

Introduction

About this Manual	1-1
Organization of the Manual	1-2
Conventions Used in this Manual	1-3
Buttons, Directories, and File Names	1-3

Get.It! Architectural Overview

High Level Architecture	2-2
Archway Internal Architecture	2-4
Archway Requests	2-6
Scripting	2-8
The Document Manager	2-10
Weblications	2-11

Introduction to Document Schemas

Definition of a Document Schema	3-1
Using Schemas in a Weblication	3-4

Tailoring Get.It!

Archway Architecture	4-1
Weblication Toolset.....	4-2
Before You Make Changes	4-3
File Structures.....	4-3
Application Definition File.....	4-4
Archway.ini Use of Packages	4-4
Presentation Folders.....	4-4
Common Components	4-5
Displaying Form Information.....	4-5
Debugging Changes.....	4-7
Where to Make the Modifications	4-8
Printing Forms	4-9
Prepare Report for Default Printer (browser directs the print spool).....	4-9
Necessary Information.....	4-11
Running the wbuild Command.....	4-11
Changing Form Contents	4-12
Adding Form Fields.....	4-12
Data for the New Field (Scripts).....	4-15
Split Frames (forms)	4-15
Localizing Your Changes.....	4-15
Adding Fields to a Document.....	4-16
Undefined Schema Fields.....	4-17
Changing Script Behavior	4-19
Changing JScript	4-19
Changing Weblication Components Layout (XSL).....	4-22
When Do I Change the XSL?	4-22

Integrating a New Product with Get.It!	4-22
Integrating a URL	4-23
Adding a URL as a Module	4-23
Adding a URL as an Activity	4-24
Adding a New Module	4-25
Adding a Feature from AssetCenter	4-26
Portal	4-27
Customizing (GUI)	4-29
Customize	4-29
Copy	4-29
Move	4-29
Hide/Show	4-29
Remove	4-30
Select Content	4-30
Get.It! Weblication Components	4-30
Personal Utilities	4-31
Create New (web page)	4-31
Change Layout	4-32
Netscape Navigator	4-33
Edit Preferences	4-34
Portal API	4-34
Portal Components	4-35
<portal-category>	4-36
<portal-component>	4-37
<plugin>	4-37
Portal Plug-Ins	4-38
Skins and Stylesheets	4-39
Directory Structure	4-41
XML Usage	4-43
XSL (example)	4-43
JSP (example)	4-43
JAVA Source Code (example)	4-44
External JavaScripts	4-44

Production Environment.....	4-44
------------------------------------	-------------

Tailoring Get.Answers!

Modifying the Site Definition File.....	5-2
------------------------------------------------	------------

Modifying the Hit List.....	5-3
------------------------------------	------------

Hiding a Hit List Column.....	5-3
-------------------------------	-----

Including the Path Name with the Title.....	5-4
---------------------------------------------	-----

Adapters

ServiceCenter Adapter.....	6-2
-----------------------------------	------------

Archway.ini Parameters.....	6-2
------------------------------------	------------

ServiceCenter Event Handling.....	6-2
-----------------------------------	-----

Using the _event parameter.....	6-3
---------------------------------	-----

Troubleshooting the ServiceCenter Database Connection.....	6-4
------------------------------------------------------------	-----

AssetCenter Adapter.....	6-5
---------------------------------	------------

INI Parameters.....	6-5
---------------------	-----

Troubleshooting the AssetCenter Database Connection.....	6-7
----------------------------------------------------------	-----

AssetCenter Feature Links.....	6-7
--------------------------------	-----

JDBC Adapter.....	6-9
--------------------------	------------

Adding a JDBC Adapter.....	6-10
----------------------------	------

Verifying the System DSN.....	6-10
-------------------------------	------

Updating the Archway.ini File.....	6-12
------------------------------------	------

Validating the JDBC Adapter Connection.....	6-13
---------------------------------------------	------

Adding the New Database Settings.....	6-14
---------------------------------------	------

Updating the Admin.xml File.....	6-14
----------------------------------	------

Creating an Interface to the Database.....	6-16
--------------------------------------------	------

Preventing Lost Connections.....	6-17
----------------------------------	------

Calling a Stored Procedure.....	6-18
---------------------------------	------

Using a URL.....	6-18
------------------	------

Using a JScript.....	6-18
----------------------	------

Using a Document Insert Or Update.....	6-19
----------------------------------------	------

LDAP Adapter.....	6-20
--------------------------	-------------

Connecting LDAP to Get.It!.....	6-20
---------------------------------	------

Updating the archway.ini file	6-21
E-mail Adapter	6-21
Verifying Adapter Connections	6-24
Log Files	6-25
Localization Support	
General Localization Steps	7-1
Archives	7-2
Externalizing ECMAScript Messages	7-2
Externalizing ECMAScript Messages with Variables	7-3
Externalizing Messages in XSL Templates	7-3
Japanese Locale	7-4
Troubleshooting	
Weblication	8-1
Weblication Reference	
Weblication Structure	A-1
Weblication Tags	A-2
<application>	A-2
<module>	A-3
<activity>	A-4
<form>	A-6
<redirect>	A-8
component	A-9
form fields	A-10
<fieldtable>	A-11
<action>	A-14
TARGET	A-16

TEXT	A-17
\$\$ (X)	A-18
<menu>	A-18
Link Attributes	A-19
<table>	A-20
Column Types	A-21
<columns>	A-23
<listbox>	A-24
<field>	A-25
<input>	A-26
<input> (Text Field)	A-27
<input> (Text Area)	A-28
<input> (Combo/Selection Box)	A-28
<input> (Checkbox)	A-29
<input> (Radio)	A-30
<input> (Hidden)	A-31
<input> (Date)	A-32
<input> (Money)	A-32
<link>	A-33
Reusable Form Components (Subforms)	A-33
Additional Tags	A-35
<html>	A-35
Additional Functionality	A-35
Searchable Popup Window	A-36
Defining New Popup Windows	A-36
 Document Schema DTD	
Document Schema Files	B-1
Schema Attributes	B-2
<document>	B-2
Nested <document> Tags	B-2
<attribute>	B-3
<collection>	B-5
ServiceCenter-Specific Attributes	B-5

JavaScript

Script Polling Manager	C-1
File Initialization Format.....	C-1
Writing A Polling Routine.....	C-2
Sample Polling Scripts.....	C-2
Read History Array	C-3
Extract URL Argument	C-3
Searchable Popup Window.....	C-4
Passing External JavaScript.....	C-5

Index

Chapter 1

Introduction



Peregrine Systems' Get.It! product suite is a line of employee self-service applications. The Get.It! applications empower employees to help themselves to functions once requiring numerous e-mails, phone calls, inter-office correspondence, and paperwork to complete. For example, the Get.Resources! application streamlines the MRO procurement cycle by drastically reducing cost and time while simultaneously increasing employee productivity and satisfaction.

Get.It! applications are accessible on the corporate intranet via web browsers. The user interface, a best of the web experience, is role-based and you can tailor it to meet your needs.

Get.It! applications benefit organizations both by freeing employees from time-consuming tasks and by automating inefficient processes such as procurement, service, and searching for answers to common questions.

About this Manual

The *Get.It! Tailoring Guide* describes the underlying architecture of Peregrine Systems' Get.It! applications and how to tailor the applications to suit your needs.

The *Get.It! Tailoring Guide* is used with several other manuals, which are:

- Operating guides, reference manuals, and other documentation for your PC hardware and operating software.
- The *Get.It! Installation Guide* which describes how to install and configure Get.It! on both a Windows and Solaris server.
- The *Get.It! Administration Guide* which describes the administration functions of Get.It! including the Administration Module and user ID maintenance.
- The Get.It! online help and documentation, located in the `...\Program Files\getit\docs` directory.
- JRun documentation located in the `...JRun\docs` directory.

To use this manual effectively, you should have a working knowledge of XML and java scripting.

Organization of the Manual

This manual is organized around the main functions associated with tailoring Get.It!. The following chart shows you which parts of the manual you need to reference to find the information you need.

To Find This	Look Here
Background information; how to use this manual.	Chapter 1: Introduction
Information about the Archway Architecture; archway requests; scripting; the Document Manager; basic information about weblications.	Chapter 2: Get.It! Architectural Overview
Introduction to document schema definitions; definition of a document schema; and using a schema in a weblication.	Chapter 3: Introduction to Document Schemas
Steps on how to tailor Get.It!; what to do before you change anything; where to save your changes; changing scripts; changing schemas; changing components of a weblication; integrating a new module into Get.It! Also explains the new portal interface and how to modify it.	Chapter 4: Tailoring
Explains how to use Get.Answers!	Chapter 5: Get.Answers!
Information about the Get.It! adapters, including instructions for connecting to a database using the JDBC adapter.	Chapter 6: Adapters
There are five compatible languages: English, French, German, Italian, and Japanese.	Chapter 7: Localization
The weblication structure; descriptions of individual elements and attributes (tags) used in documents; reusable form components.	Appendix A: Weblication Reference
Document schema files; schema attributes; tags you can use in schemas.	Appendix B: Document Schema DTD

Conventions Used in this Manual

Most screen shots in this manual come from the Windows version of Get.It!. The action you should take on the window is usually explained in the step below the sample. Pay special attention to text, or notes, next to a screen shot. For example:

Note: Make sure your form statistics are displayed. See “Showing Form Information” on page 4-3 for instructions.

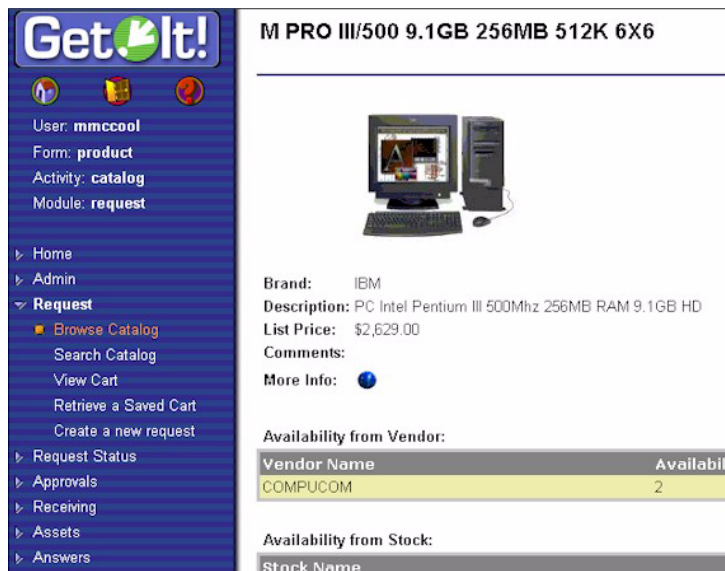


Fig. 1.1 Adding a Field to a Form.

All screens in this manual are for sample purposes only. They are shown in the Classic stylesheet.

Buttons, Directories, and File Names

The following conventions are used when describing buttons on the windows, paths for directories, and file names.

- Buttons you click on are shown in bold such as “Click **Next**.”

- Directory paths are shown in Courier New font, such as `C:\Program Files\getit\`. The directories used in this manual are the default directories assigned during the installation. If you change the directory into which you install Get.It! or JRun, take note of the correct directory and replace the default path with the one that is correct for your system.
- File names are shown in Courier New font, such as `login.asp`.
- When showing XML codes in the samples, “...” is often used to signify that some of the lines have been removed because they are unnecessary to the topic. Samples of code are not entire files but they are representative of the information being discussed in that section.



Chapter 2

Get.It! Architectural Overview

This document introduces the architecture behind Get.It!, Peregrine Systems' product suite that includes applications such as Get.Resources!, Get.Answers!, and Get.Service! The Get.It! suite is built on top of the Archway architecture. This architecture offers a simple and extensible way of creating new applications and interfacing with Peregrine Systems existing systems, including AssetCenter and ServiceCenter.

The architecture has been designed with specific goals:

- Offer services to everyone in an organization
- Offer access everywhere users need it
- Offer support related to everything in the infrastructure that helps employees get things done

These goals mean that the Get.It! architecture is designed to make services available to users through common interfaces like web browsers, handheld computing, and even mobile phones. The applications are designed to provide a wide range of services, from helping a user with a PC problem, to allowing the creation of a purchase request, to reporting a problem with the employee's office space. Peregrine Systems Infrastructure Management applications offer many of these services, and the Get.It! suite makes the services available to everyone, everywhere.

High Level Architecture

Get.It! applications and interfaces are implemented using basic building blocks that include:

HTTP	A simple and widely supported protocol for sending client requests to a server. Variations such as HTTPS provide security as well.
XML	This rising technology is a very natural way to represent data rich documents.
Commercial web servers	The services provided by the Archway architecture can be served from any commercial web server, including IIS, Apache, Netscape Enterprise Server, or the Java Web Server.
Common clients	Applications can be built to be deployed via web browsers (IE, Netscape), handheld devices (Palm Pilot), or mobile phones (through HDML).

The following diagram illustrates the architecture:

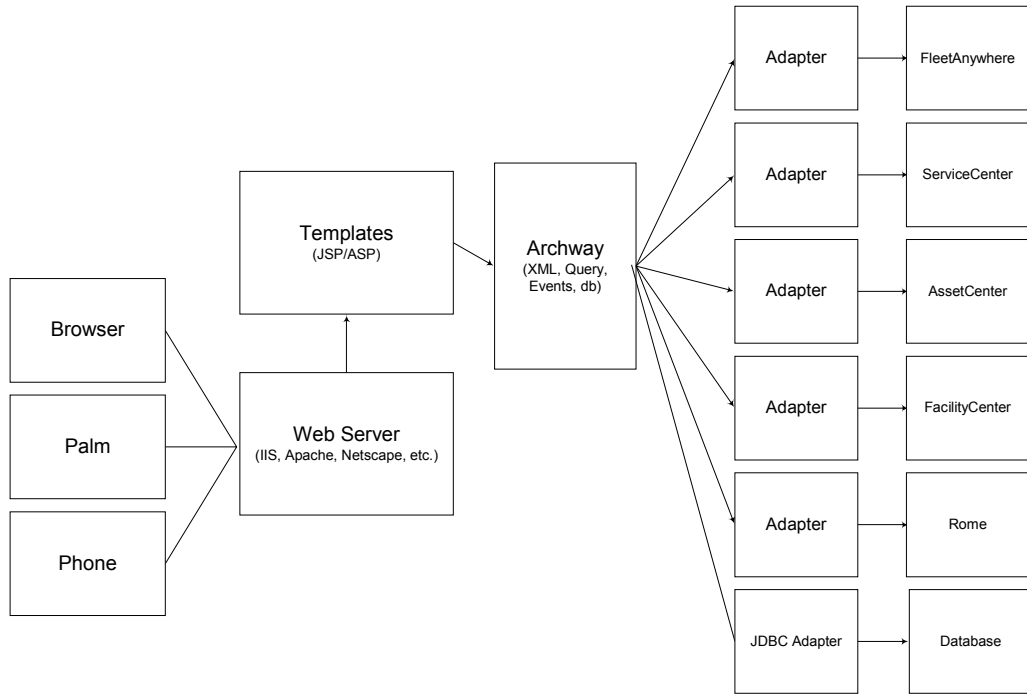


Fig. 2.1 The architecture

At the center of the architecture is a component named Archway. This component is designed for a simple purpose: it listens to HTTP requests from arbitrary clients, routes the requests to an appropriate server, and returns data or documents. The requests supported by Archway can vary, but they fundamentally consist of queries, data updates, or system events.

For example, a client can contact Archway and ask to query ServiceCenter for a list of tickets. Another client could contact Archway and supply it with a new purchase request that should be entered into AssetCenter's database. Yet another client could contact Archway to open a new problem ticket through an Event Services event (e.g., a PMO).

All requests and responses are formatted using XML (Extensible Markup Language). XML provides a human readable self-describing syntax for defining documents. For example, a problem ticket expressed in XML could appear as follows:

```
<problem>
  <number> PM5670 </number>
  <contact> Joe Smith </contact>
  <description> My printer is out of paper </description>
</problem>
```

Clients that interact with Archway can do anything they need with the XML that is returned as a response. Very frequently, the client initiating the request is a user interface such as a web browser. Such a client could easily display the XML documents returned by Archway. However, to be of better use, the XML documents are often displayed within a formatted HTML page. This is accomplished by using popular and commercially supported technologies such as Microsoft's ASP (Active Server Pages) or Java's JSP (Java Sever Pages).

Both JSP and ASP provide a syntax for creating HTML pages that is pre-processed by the web server before being sent to the browser. During this processing, XML data obtained from Archway is merged into the HTML page. Later in this document we introduce the related concept of a weblication. A weblication is a term used to refer to an application running on the web. Archway's architecture includes special support for automatically generating the pages (i.e. HTML, JSP) that make up a weblication.

Archway Internal Architecture

The internal design of Archway is simple and very flexible. Archway is implemented as a Java servlet—a Java application that is executed by a web server. HTTP requests sent to the web server are forwarded to the Archway servlet for processing. When the processing is done, the web server returns the output generated by Archway.

Each request is interpreted to determine its destination. Specifically, Archway is able to communicate with a variety of back-end systems like AssetCenter or ServiceCenter. Requests can be handled in one of three ways:

1. A request can be sent directly to an adapter that talks to a back-end server. For instance, a query request for opened tickets could be forwarded to an adapter capable of communicating with ServiceCenter.
2. A request can be sent to a script interpreter hosted by Archway. This is a very powerful feature. It allows Peregrine Systems and customer developers to define their own application specific services. Within a script, calls can be made back to Archway to access the back-end system with database operations and events.
3. Finally, a request can be sent to a component known as a Document Manager. This component provides automated services for combining logical documents.

The following diagram illustrates the internal Archway architecture.

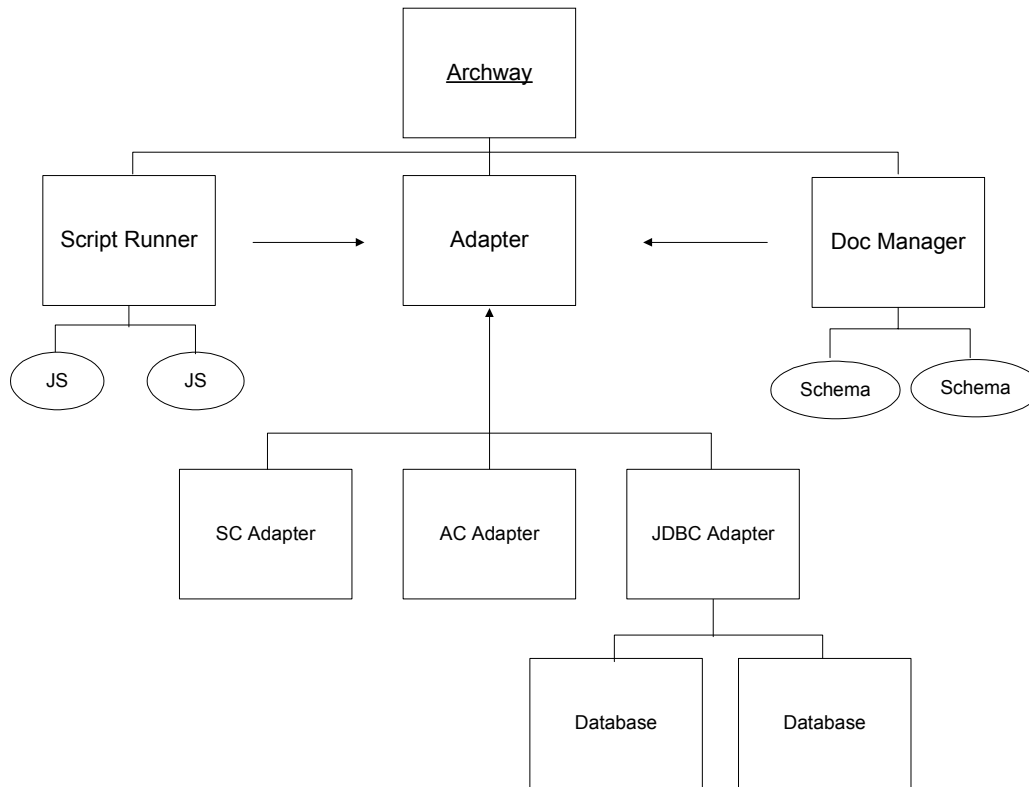


Fig. 2.2 Archway's internal architecture

Archway communicates with back-end systems with the help of specialized adapters that support a predefined set of interfaces for performing connections, database operations, events, and authentication. All adapters use DLLs to communicate with each application.

Messages can be routed to a script interpreter hosted by Archway. The interpreter supports ECMAScript. JavaScript (Netscape) and JScript (Microsoft) were both created based upon the original ECMAScript language.

Messages can be routed to the Document Manager component. This component reads special schema definitions that describe application documents for logical entities such as a Purchase Request, Problem Ticket, or Product Catalog. The Document Manager uses these schemas to automatically generate database operations that query, insert, or update such documents.

Archway Requests

Archway supports a variety of requests, all of which are based on two basic technologies: HTTP and XML. The HTTP protocol defines a simple way for clients to request data from a server. The requests are stateless and a client/server connection is maintained only during the duration of the request. All this brings several advantages to Archway, including the ability to support a large load of requests with the help of any of today's commercial Web Servers.

Another important advantage is that any system capable of making HTTP requests can contact Archway. This includes web Browsers, of course. But in addition, all modern programming environments support HTTP. This makes it very simple to write new adapters that communicate with Peregrine Systems servers without the need of specialized APIs.

From a simple point of view, an HTTP connection consists of:

- A client request
- A server response

The messages exchanged normally have a number of header lines and some content lines. For this discussion, let's focus on two principal parts of a request:

Query String	This represents the <i>parameters</i> sent along with the URL for the HTTP connection. For instance, consider the following URL: http://prgn/archway?hello&world . This URL is made up of a server locator (http://prgn/archway) and a query string (hello&world).
Content	A request can also include an arbitrary amount of data appended to the request. This data could follow any format, but for Archway, the data is always formatted as XML.

Archway uses the query string of a request to determine what it has been asked to do. The following query string syntax is expected:

[archway?target.command¶m=value¶m=value&...](#)

Let's consider each part of the request.

Target	The name of a target object that should handle the request. Remember that Archway's job is to forward requests to a system and return the response. Thus, the target could be ServiceCenter, AssetCenter, etc. As we will see, the target may also be the name of a Script Object that contains customizable logic for handling the request.
Command	The command describes the action that the target object should take. By default, there are five basic actions that may be supported: query, update, insert, delete, and event. However, when the target is a Script Object, the action can be any function defined by the script.
Param=Value	An arbitrary number of parameters can be passed along with the request. The encoding of these parameters is the same as that used by CGI (the Common Gateway Interface). This makes it seamless to make Archway calls from a web page. As with CGI, data sent by a browser is provided by fields embedded in an HTML form. This data is automatically formatted as a CGI request in a way that Archway understands.

The following are some sample URLs that illustrate the power of contacting Archway with HTTP requests that return XML documents. These samples are intended as an introduction.

```
archway?sc.query&_table=probsummary&priority.code=1
```

This sends a query request to ServiceCenter for all records in the probsummary table with a priority code of 1.

```
archway?ac.query&_table=amProduct&_return=Brand;mPrice;Model&count=2
```

This sends a *query* request to AssetCenter for the first two records in the *amProduct* table. Only the **Brand**, **mPrice**, and **Model** fields are returned for each record.

```
archway?sc.pmo&contact.name=David+Baron&$ax.field.name=This+is+a+demo
```

The sample above creates a new ticket in SC by sending a *pmo* request with two parameters.

```
archway?test.helloWorld&greeting=Hello
```

This sample sends a *helloWorld* request to a script object named **test**.

You could try URLs like these from a web browser to see first hand how the Archway requests work. The figure below illustrates this by showing the XML results of a query for

products from AssetCenter.

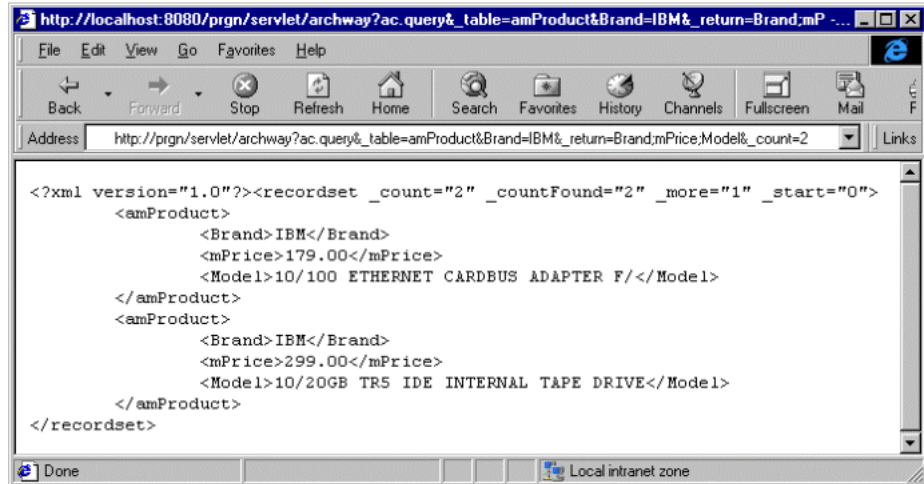


Fig. 2.3 Testing URLs from a web browser

Scripting

A great deal of Archway's flexibility and power comes from its support of the ECMAScript language. This enables application developers to define arbitrary code that handles client requests. ECMAScript is a standard version of the language originally made popular by Netscape (JavaScript) and later adopted by Microsoft (JScript).

ECMAScript is a very powerful language, but it allows for simple tasks to be accomplished in a simple manner. Its syntax is similar to that of Java, and yet traditional JavaScript is not Java. While this is true, one interesting aspect of Archway's ECMA support is that it includes the ability to access any arbitrary Java object. This makes Archway scripts even more powerful since they have all the power of Java available within the easy programming syntax of ECMAScript.

It is beyond the scope of this document to describe all aspects of ECMAScripting. One reason for adopting this language is that it is standard, well known, and widely documented. Numerous references and guides exist for the language. To start, the ECMA web site can be located at <http://www.ecma.ch>. A good book with very comprehensive language description and references is *JavaScript - The Definitive Guide* by David Flanagan (O'Reilly).

ECMAScript, JavaScript, and JScript tend to vary in some way or another. This is especially true in the APIs for what is known as *Client Side JavaScript*. This is the type of scripting supported by a browser to allow dynamic manipulation of what gets displayed

within a web page. However, none of this really matters in the context of Archway. Archway uses what is known as *Core JavaScript*. This is the subset of the language that is independent of any client side (Browser) features. Archway executes all script code on the server while processing a request. When the script is done executing, its response is sent back as XML to the client.

Much of the ability to write useful scripts comes from a very small set of *Scriptable* Objects that are supplied with the Archway architecture. Two of the main objects provided are:

Messenger	This object allows any script to send messages back to Archway. For example, through the messenger, a script can ask Archway to send a query to AssetCenter or an event to ServiceCenter.
Message	This object encapsulates XML documents in a very easy to use API. With this object, scripts can very easily build and interpret complex XML documents.

Below is a sample ECMAScript that illustrates the ease of programming provided by these objects. The script executes a query against AssetCenter:

```
function getCatalog( msg )
{
    var msgProducts;
    msgProducts = archway.sendQuery(
        "ac", "SELECT Brand,mPrice FROM AmProduct", 0, 10 );
    return msgProducts;
}
```

Here is another sample script that sends a PMO event to ServiceCenter:

```
function getCatalog( msg )
{
    var msgEvent;
    var msgResponse;

    msgEvent = new Message( "pmo" );
    msgEvent.set( "contact.name", msg.get("UserName" ) );
    msgEvent.set( $ax.field.name, msg.get( "Description" ) );
    msgResponse = archway.sendEvent( "sc", msgEvent );

    return msgResponse;
}
```

These examples are shown to demonstrate the basic concept of scripting in Archway. Details on script design and the Object interfaces that they may use are documented in Chapter 3, "Introduction to Document Schemas."

The Document Manager

The Archway uses XML to exchange data and documents between clients and the supported back-end systems. Fundamentally, the XML data returned by Archway is obtained by executing queries against one or more systems. The queries could be executed by a direct URL request or indirectly within an ECMAScript.

Simple queries are only capable of returning record sets of data. However, clients are more often interested in exchanging documents. A Document is a logical entity built up of several pieces of data that can come from various physical database sources. For example, consider a Product document. Products have a number of individual fields such as **Price** or **Brand**. They also may have collections of other related documents, such as a collection of *Vendors*. Below is sample XML for a Product document:

```
<product>
  <brand> IBM </brand>
  <model> ThinkPad 770 </model>
  <price> 1250 </price>
  <vendors>
    <vendor>
      <name> Best Buy </name>
      <phone> 267-8967 </phone>
    </vendor>
    <vendor>
      <name> Super City </name>
      <phone> 267-8967 </phone>
    </vendor>
  </vendors>
</product>
```

Building such a Product document can certainly be accomplished by running several queries and putting the results together in an XML message. An ECMAScript is a perfect place to code such logic.

However, there is an even better way to build documents with the use of Archway's Document Manager. This component provides the very important service of processing logical Document Schema Definitions and automatically generating queries or database operations to create and process these documents.

Here is a small example of a document schema that defines what Product documents should look like:

```
<document name="Product">
  <attribute name="Id" type="num"/>
  <attribute name="Brand" type="string"/>
  <attribute name="Model" type="string"/>
  <attribute name="Price" type="money"/>
```



```

<collection name="Suppliers">
  <document name="Supplier">
    <attribute name="Name" type="string"/>
    <attribute name="Phone" type="string"/>
  </document>
</collection>
</document>

```

Note: The principal concept to notice is that a document schema describes the fields and collections that make up a document. The details on how to construct document schemas are documented in Chapter 3, "Introduction to Document Schemas."

The Document Manager can be accessed with direct URL calls to Archway, as well as from ECMAScripts. Here is a sample script that retrieves Product documents:

```

function Product( msg )
{
  return archway.sendDocQuery( "ac", "Product", msg );
}

```

Weblications

So far we've described architecture components that make up the plumbing of Get.It! applications. If an application is to be deployed on a web Browser, there remains one piece that must be defined to create the application: the screens and the flow for navigating among them.

Web Browsers display screens defined in HTML. The screens can contain data retrieved from the server, and they may also provide entry fields for sending input data back to the server.

To understand how Archway fits in with the creation of browser interfaces, let's start by considering the example of setting up a web page that lets a user create a new Service-Center ticket. Defining this page in HTML might appear as follows:

```

<form action="http://prgn/archway?sc.pmo" method="GET">
<input type="text" name="contact.name"> <br>
<input type="text" name="$ax.field.name"> <br>
<input type="submit" value="Open"/>
</form>

```

Even if you are not familiar with HTML, the code above should be simple to understand. The first line defines an HTML form. All forms have an action property that tells the browser where to send the data typed in by the user. In this case, we see that data will be sent to <http://prgn/archway?sc.pmo>. The next two lines contain input fields, each associated with a named field: `contact.name` and `$ax.field.name`.

In essence, the HTML above sends a pmo message to ServiceCenter through Archway. The data typed into the entry fields are passed in as PMO parameters.

What about using HTML to display data retrieved via Archway? As mentioned earlier, Archway is designed to return XML documents that can be merged into an HTML page using technologies such as JSP or ASP. Below is a sample snippet of JSP that sends Archway a query for ServiceCenter tickets and displays the results in an HTML table.

```
<html>
  <table>
  <%
Message msg = messenger.sendQuery(
  "SELECT number,brief.description FROM probsummary" );
List list = msg.getList( "probsummary" );
for ( int iCurrent = 0; iCurrent < list.getLength(); iCurrent++ )
{
  %>
  <tr>
    <td> <%= list.get( iCurrent, "number" ) %> </td>
    <td> <%= list.get( iCurrent, "brief.description" ) %> </td>
  </tr>
  <%
  }
  %>
  </table>
</html>
```

The code above is basically an HTML page with Java code mixed in. The Java code uses a few objects defined by Archway. These are shown in bold, and they include a messenger that talks to Archway, a message class that encapsulates XML responses, and a list object that allows easy navigation of a result set.

While these two samples of code are interesting to understand, it is not necessary to learn much if anything about HTML, JSP, or ASP development to write a Weblication with Archway. This is because Archway provides some additional tools that automatically generate the underlying HTML and JSP code that makes up an application.

Before introducing these tools consider the following. Below is a screenshot of a page in

Get.Resources. The page shows a table with results from a catalog search.

The screenshot shows the Get.It! web application interface. On the left is a navigation menu with categories like Home, Admin, Request, Browse Catalog, Search Catalog, View Cart, etc. The main content area is titled 'Search Results' and contains a table of search results. The table has columns for Count, Brand, Model, Description, and Price. Below the table are buttons for 'Add to shopping cart' and 'Go Back'. Annotations with arrows point to various parts of the interface: 'Application Title' points to the Get.It! logo; 'Activities' points to the navigation menu; 'Actions' points to the bottom buttons; 'Form Title' points to the 'Search Results' header; and 'Form Contents' points to the table of search results.

Count	Brand	Model	Description	Price
1	IBM	PC300GL PIII450 BX 4.2GB 64MB W98	PC Intel Pentium III 450MHz 64MB RAM 4.2GB HD	\$1,119.00
1	IBM	PC300GL PIII500 BX 8.4GB 64MB W98		\$1,299.00
1	IBM	PC300PL PIII450 64MB 13.5GB HD		\$1,429.00
1	IBM	PC300PL PIII500 64MB 13.5GB HD		\$1,499.00
1	IBM	PC300PL PIII500 512KB 6.4GB 64MB		\$1,329.00
1	IBM	M PRO III500 9.1GB 256MB 512K 6/6	PC Intel Pentium III 500MHz 256MB RAM 9.1GB HD	\$2,629.00
1	IBM	MPRO INTELLISTATION 2D 2-PIII550		\$3,819.00
1	IBM	PC300PL PIII500MT 13.5GB 64MB 40X		\$1,579.00
1	IBM	INTELLISTATION EPRO PIII500 13.5GB		\$2,205.00
1	HEWLETT PACKARD	KAYAK XA MT PIII500 9.1GB 128GB		\$2,332.00

Fig. 2.4 Sample Get.It! Weblication window

This page is part of a Get.It! weblication. As such, it conforms to a predefined template that determines a regular layout and placement of several components within a browser page. The actual template is customizable, and therefore not all Weblications have to look as the one pictured above.

The creation of the page above is made possible by three ingredients:

1. XSL Layout templates - These templates define the layout and organization of items in a web page. They are defined using the Extensible Stylesheet Language (XSL). This is an XML based language that is becoming more and more widely used to format Web pages out of XML data. Out of the box, Peregrine Systems may supply one or more XSL templates. Customers could choose among templates in a similar way that tools like Microsoft Word or Powerpoint allow writers to choose from predefined document types or templates.

To learn more about XSL, you can consult the W3C web site. Their XSL specification is found at <http://www.w3.org/TR/1999/WD-xs1t-19990421.html>. Microsoft also posts information on XSL at <http://msdn.microsoft.com/xml/xslguide/>.

Note: Microsoft's XSL support varies slightly from the W3C specs. Therefore, consult the W3C specs for the most accurate information. Because Archway's support for XSL is implemented on the server, browser support is not necessary or relevant.

2. Cascading Style Sheets (CSS) - All aesthetic aspects of a web page are defined separately in a CSS file. This includes specifications for colors, fonts, alignment rules, and even some special effects.
3. Weblication Definition - The actual application specific portions of a weblication are defined using a concise high level XML description.

Defining weblications in this manner has several advantages. First, it is much simpler than having to hand code numerous HTML and JSP pages. Second, it makes it easy to define a consistent look and feel to a web site. A simple change to a template is quickly propagated to what could be hundreds of page files. Finally, the pages created automatically for a weblication include a number of features to deal with user authentication, security, access rights, and session tracking.

To illustrate this further, here is the XML weblication description for the form displayed above:

```
<form name="catalog" onload="procure.getCatalog">
  <title> Get.Desktop </title>
  <instructions>
    Here are the items found in this category. You may click on any
    one to see a detailed description, or you may simply enter a
    count to add items to your order.
  </instructions>
  <table record="Product" rows="10">
    <link target-form="product" field="Id"/>
    <column label="Count"      field="nCount" type="select"/>
    <column label="Brand"      field="Brand"/>
    <column label="Model"      field="Model"/>
    <column label="Price"      field="Price"/>
  </table>
  <actions target-activity="review">
    <submit name="bTable"> Add to shopping cart </submit>
    <back/>
  </actions>
</form>
```

Just this small amount of XML is responsible for almost the entire window in figure 2.4. An Archway tool parses this definition and generates the necessary HTML and JSP code that creates proper input to the browser. Compare this to the JSP and HTML code shown at the top of this section, and it is quickly evident that the weblication approach provides a much simpler way to define applications.

There are a few things worth highlighting in this XML definition. First notice that the form has an `onload` property. It specifies that when constructing the page, an Archway script named `getCatalog` should be invoked. This script is defined to return Product documents.

For instance, each product could have the following XML definition:

```
<Product>
  <Brand> X </Brand>
  <Model> Y </Model>
```

```
<Price> Z </Price>  
</Product>
```

This XML data is easily incorporated into the HTML page. The form defines a table element that references fields in the XML Product description, and Archway takes care of generating the proper code to extract the fields from the XML documents.

Again, this is just an introduction to the concept of a weblication. This is probably the most important part of the Archway architecture to understand because it is directly related to the ability to customize the Get.It! applications or to define new ones. Details on the weblication definition language are documented in Appendix A, "Weblication Reference." In addition, related information can be found in Chapter 4, "Tailoring Get.It!."



Chapter 3

Introduction to Document Schemas

The Archway Document class provides Archway weblications and scripts with the service of processing logical Document Schema Definitions and implementing the physical database access operations for querying and constructing documents.

For example, consider a "Product" document. Products have a number of individual fields such as "Price" and "Brand." They also have collections of other sub-documents such as a collection of "Vendors."

The queries that create a document vary depending on the physical schema of the system hosting the "Product" data (such as AssetCenter and ServiceCenter). To understand how to construct these queries, the class reads an XML "Document Type Definition" (DTD) file.

The DTD file contains *Base Document Definitions* that define the fields, collections, and nested documents that make up a logical Document.

In addition, the DTD file defines *Derived Document Definitions* with physical database schema information for building a base document out of data found in a specific system (such as AssetCenter and ServiceCenter). A Derived Document Definition may define physical table and field information for some (but not all) of the fields in a Base Document.

Definition of a Document Schema

A document is defined as a collection of one or more *Attributes*. Each attribute is a "field" in the document. For example, a *Product* document may have a *Price* attribute, zero or more nested *Documents*. This allows documents to be nested inside each other recursively for zero or more nested *Collections*. A collection is a Document attribute which in turn has a list of one or more nested documents. For instance, a *Product* may have a *Suppliers* collection with one or more *Supplier* documents.

The following is an example that demonstrates most elements of the XML schema for defining documents:

```
<documents name="base">
  <!-- Product Document -->
```

```

<document name="Product">
  <attribute name="Id"                type="num"/>
  <attribute name="Brand"             type="string"/>
  <attribute name="Model"             type="string"/>
  <attribute name="Price"             type="money"/>

  <!-- Here is an example of a nested document reference -->
  <collection name="Suppliers">
    <document name="Supplier"/>
  </collection>
</document>

<!-- Supplier Document -->
<document name="Supplier">
  <attribute name="Name"              type="string"/>
  <attribute name="Price"             type="money"/>
</document>

</documents>

```

The following is a more complete sample XML DTD of a Product document. The sample shows some additional, important concepts such as:

- Schemas are organized into "base" and "derived" versions. In the examples that follow, the first is a base schema and the second is a derived schema.
- Derived (system specific) schemas map to a specific system and are used to generate queries. A derived schema must map to a system from which the information will be accessed.
- Nested documents can be defined in place or as references.


```

<?xml version="1.0"?>

<!--=====
Name:   schema.xml
Author: David Baron
Date:   10/99
=====-->

<schema>

<!--=====
Generic Schema Definitions
=====-->

<documents name="base">

  <!-- Product Document -->
  <document name="Product">

    <attribute name="Id"           type="num"/>
    <attribute name="Certification" type="string"/>
    <attribute name="Category"     type="string"/>
    <attribute name="Brand"        type="string"/>
    <attribute name="Model"        type="string"/>
    <attribute name="Comment"      type="string"/>
    <attribute name="Price"        type="money"/>
    <attribute name="Description"  type="string"/>
    <attribute name="PhotoId"      type="number"/>
    <attribute name="IconId"       type="number"/>

    <!-- Here is an example of a nested document reference -->
    <collection name="Suppliers">
      <document name="Supplier"/>
    </collection>

    <!-- Here is an example of a nested document definition -->
    <collection name="Stocks">
      <document name="Stock">
        <attribute name="Name"       type="string"/>
        <attribute name="Quantity"   type="string"/>
      </document>
    </collection>

  </document>

  <!-- Supplier Document -->
  <document name="Supplier">
    <attribute name="Name"           type="string"/>
    <attribute name="Price"          type="money"/>
    <attribute name="Delivery"        type="time"/>
    <attribute name="Available"      type="number"/>
    <attribute name="URL"            type="url"/>
  </document>

  <!-- Catalog Document -->
  <document name="catalog">
    <collection name="Products">
      <document name="Product"/>
    </collection>
  </document>

```

In a "base" schema the document is defined within the schema.

This is an example of a referenced nested document.

This is an example of a nested document in place.

In a derived schema, the document is created by information which is accessed from another system. In this example, the data will be accessed in the "amProduct" table from within AssetCenter.

```

<!--=====
AssetCenter Schema Derivations
=====-->

<documents name="ac">

  <!-- AC Product Document -->
  <document name="Product"          table="amProduct">

    <attribute name="Id"              path="lProdId"/>
    <attribute name="Catergory"       path="Category.Name"/>
    <attribute name="Comment"        path="Comment.memComment"/>
    <attribute name="Price"          path="mPrice"/>
    <attribute name="PhotoId"        path="lPhotoId"/>
    <attribute name="IconId"         path="lIconId"/>
    <attribute name="Description"    path="cf_Description"/>

    <collection name="Stocks">
      <document name="Stock"          table="amProdStockLine">
        <attribute name="Name"        path="Stock.Name"/>
        <attribute name="Quantity"    path="lTotalQty"/>
      </document>
    </collection>

  </document>

  <!-- Supplier Document -->
  <document name="Supplier"         table="amProdSupp">
    <attribute name="Name"           path="Supplier.Name"/>
    <attribute name="Price"          path="mPrice"/>
    <attribute name="Delivery"       path="tsDelivDelay"/>
    <attribute name="Available"     path="lQtyAvail"/>
    <attribute name="URL"            path="Product.fv_ManufacturerURL"/>
  </document>

</documents>

<!--=====
Done
=====-->
</schema>

```

Using Schemas in a Weblication

In using Document and Schema support, "document" type archway messages are available to ECMAScripts. Here is a script that queries for a list of Product documents (sendDocQuery):

```

function getCatalog( msg )
{

```

```
return archway.sendDocQuery( "ac", "Product", msg );
}
```

The DocumentManager also supports SQL-like queries. For instance, you can query as in the following example:

```
archway.sendDocQuery( "ac", "SELECT Brand,Description FROM Product
WHERE Category='Desktop'ORDER BY Brand", 0, -1 );
```

You can also accomplish Document querying in the following manner:

```
msgParam.set( "_return", "Brand;Description" );
msgParam.set( "Category", "Desktop" );
msgParam.set( "_sort", "Brand" );
archway.sendDocQuery( "ac", "Product", msgParam, 0, -1 );
```

Use the SQL queries sparingly, especially in a weblication, because this method defeats one of the main purposes for setting up the DocumentManager. In a weblication setting you do not want hard-coded queries in your scripts. All fields that go in the `msgParam` are served for us by the weblication forms. This makes tailoring much easier. However, for certain script situations, the new syntax offers some coding comfort. Other calls include `sendDocInsert` and `sendDocUpdate`. See the Messenger API for details.

The `document` object works together with data provided by `wbuild` to do the following:

- Automatically create all queries that comprise a document.
- Use parameters passed into a script to filter the resulting Document result set. For instance, to search for Products with a particular Brand, Model, or Certification, the calling weblication needs a form with Brand, Model, or Certification fields. These are automatically added to the query if they are applicable to the document search.
- While `wbuild` generates forms from an XML weblication, it builds a list of document fields used by the form. This list is passed to the document search, allowing the Document class to limit the queries to those fields that will be used. This is very significant as it can eliminate the need for numerous sub-queries.



Chapter 4

Tailoring Get.It!

The Get.It! applications provided by Peregrine Systems are designed to be functional out-of-the-box. However, you may want to customize and tailor the applications to better fit your company's needs.

You can tailor Get.It! to do almost anything you need. The types of tailoring include things such as:

- Changing the wording or labels in a form
- Adding or removing fields on a form
- Adding fields to the Documents exchanged with the system
- Changing the behavior of a script
- Changing the layout of a weblication
- Adding or removing modules
- Translating your modifications into the supported languages

This chapter describes how to customize individual features of the weblications appearance and performance. It guides you through these different scenarios and provides several examples.

Archway Architecture

The Archway architecture is designed to accommodate the types of tailoring mentioned above.

Before you tailor Get.It!, we highly recommend you have an understanding of the archway architecture. See “Get.It! Architectural Overview” on page 2-1 for explanations of several concepts and terms that are used throughout this chapter. For a description of the weblication tags, see Appendix A, "Weblication Reference," at the back of this manual.

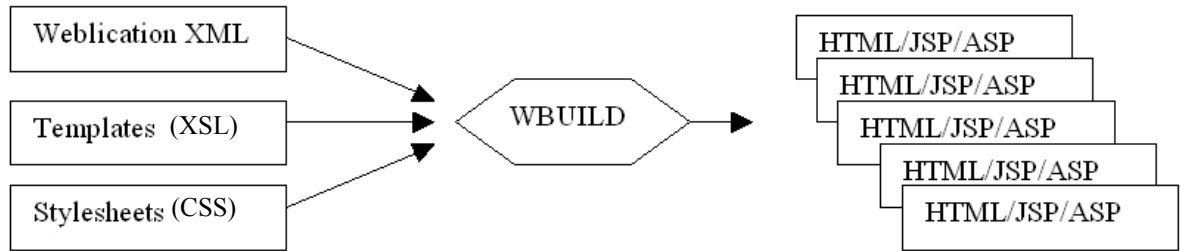
Weblication Toolset

Before you make changes to the weblication, use the Admin Module to set "Debug scripting" in the General Execution Settings and "Show form info" in the Weblication Settings to true.

Before doing any customization, you may want to review the various ingredients that make up a weblication. See Chapter 2, "Get.It! Architectural Overview," for an introduction to weblications.

Weblication XML Definitions	The XML files that define application modules, activities, and forms.
Archway ECMA Scripts	ECMA script files that implement application specific behavior.
Document Schema Definitions	The XML definitions that describe the data that should be queried or updated to create XML documents that can be interchanged with Archway by a client such as a Weblication.
Stylesheets	The colors and fonts used for pages in a Weblication.
Layout Templates	Templates that define the layout and component construction rules for creating pages in a Weblication.
WBUILD	The executable tool used to create a Weblication.

The components listed here play different roles in the overall weblication definition. The deployment of a weblication requires a compilation step that takes all of the ingredients and generates a set of web pages that are installed into a web server directory:



When you make changes to a Weblication definition, you need to re-generate the web pages by running `wbuild application` from a command prompt at the directory `...getit/bin/`.

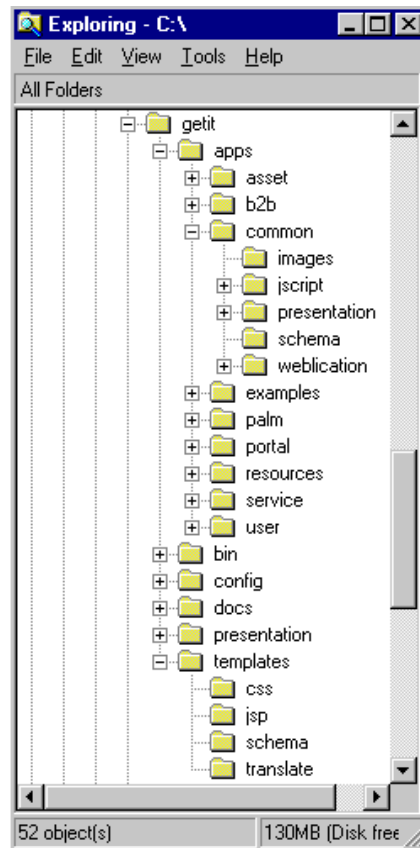
Fig. 4.1 The Weblication toolset at work

Before You Make Changes

Since the source for the weblication is provided with the product, you can make any changes you want to the Get.It! weblication. Before you start to modify Get.It! there are a few items you will need to know and set regardless of the change you are making. These tips make the process of modifying Get.It! much easier.

File Structures

The files that are used to build the Get.It! weblication are stored in the `...getit\apps` directory. The XML files, schemas, and scripts used to make up the weblication page are grouped by application within this directory into packages. These packages make it easy for you to determine which files, schemas, and scripts are used for each application.



Each package contains the XML files, schemas, and scripts that make up the application. The *common* folder holds the files that are used in more than one application.

The following folders are created for each application package. Some packages also contain folders that are specific just to that application.

- `jscript` = The script files for the application.
- `presentation` = Common browser-side presentation files. Most presentation files are generated by the `wbuild` command and are not stored in source control.
- `schema` = The schema files for the application.
- `weblication` = The XML files that make up the application.

Store all changes you make in the `.../getit/apps/user/` folder. Changes to the XML files should be stored in the *weblication* folder within the user directory, and changes to schemas should be stored in the *schema* folder.

Application Definition File

For Solaris, the file name is `getservice.xml`.

The application in this file is what you enter as the parameter in the `wbuild` command.

Applications are defined within a file that lists the packages that should be included for deployment. All applications are pulled together in the `getit.xml` file:

```
<application name="e" home="portal">
  <title> Get.It! </title>

  <modules>

    <!--=====
    List of packages imported by this deployment configuration.
    The "contents" attribute is optional: when specified it helps
    determine the order in which content module files are included. When
    not specified, all files found in a package folder are included in
    alphabetical order.
    =====>
    <package name="common" />
    <package
      name="resources" contents="request;status;approve;receive"/>
    <package name="asset" />
    <package name="service" />
    <package name="b2b" />
    <package name="portal" />
  </modules>
```

Archway.ini Use of Packages

Archway requires a setting in the `archway.ini` file in order to support packages:

```
weblication=getit.xml
```

The “weblication” parameter tells Archway which application has been deployed. At run time, Archway reads the application definition file to know which packages are included in the execution. This is important because it is used by Archway to locate scripts and schemas.

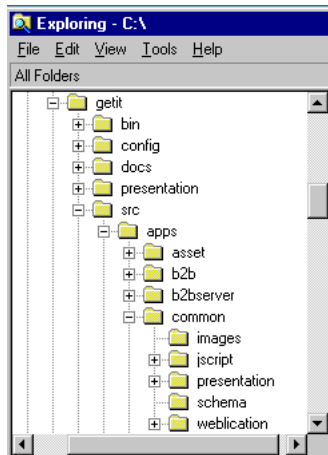
A new method, `Archway.getPackages()` is now available for run-time information about packages.

Presentation Folders

The presentation folder is a placeholder that contains no checked-in files. This folder should only contain generated files. You can define presentation folders with specific files and images, that are not automatically generated, in the individual packages. `Wbuild` pulls all presentation files into the top-level `presentation` folder. You should never check in any files in this top level folder.

Common Components

Some of the images, forms, schemas, and XML files are used more than once within the Get.It! weblication. These common components of the weblication are stored in the `...getit\apps\common` directory.



The *common* folder holds the images, scripts, presentation files, schemas, and XML files that are used by more than one application package.

Fig. 4.2 The common folder.

Displaying Form Information

In a weblication, a form contains detail fields for the product, including the model, brand, list price, etc. We have created an option in the Settings activity in the Administration Module that allows you to display the information you can use to find the form you want to change.

If “Show form info” is set to true, three lines of information are displayed on the left of the window as shown in the following sample:



Fig. 4.3 Showing the forms information

Note: **Module** = The name of the XML file in the `...getit\apps\` directory.
Activity/Form = Use these as search criteria to locate the exact form you want to change. The portal home page does not show form statistics.

Use the form statistics to determine the XML file you want to use in your modification, and to search this file to find the exact form to change.

The “module” tells us which file to edit. Look in `...\.getit\apps\application-package\weblication\. In this sample above, the file is login.xml. Within that file, look for <module name="login">. The following is the actual line you will find:`

```
<module name="login" access="anonymous" appmenu="false"
apphead="false">
```

The “Activity” tells us which activity to look for within that module. Search for `<activity name="main">`. The following is the actual line you will find:

```
<activity name="main">
```

The “form” tells us which form to edit within the activity, and the module. Search for `<form name="start">`. The following is the actual line you will find:

```
<form name="start" onload="login.init">
```

If what you want to change is the title of the module or activity, however, the form information you find is not what you want to change. To find where you would change the titles, try the following method:

For any given `<module>.xml`, look at the lines immediately following the `<module name="...">` declaration at the beginning of the file and change the data there. Look at the `request.xml` file.

```
<module name="request" access="getit.requester">
  <title> Request Menu </title>
  <description
    image="images/order.gif"
    short="Resources"
    long="Request supplies, hardware, and software" />
```

What each line tells us:

```
<module name="request" access="getit.requester">
```

This tells us which named `amUserRight` (or `ServiceCenter` capability word, if you had that) the user needs in order to access this module.

```
<title> Request Menu </title>
```

Data to show on the left hand side (in the activities) within the module.

```
<description
  image="images/order.gif"
```

Identifies the graphic to show on the main menu.

```
  short="Resources"
```

Name to use in the menu bar and on the main menu.

```
  long="Request supplies, hardware, and software" />
```

Balloon help when you hover over the short name in the menu bar.

You can change the names of the module or activity in this area of any XML file.

Now that you know how to find the form you want to modify, let's see how you ensure the changes you make are not removed when you apply a future release of Get.It!

Debugging Changes

We recommend you set all "debug" options in the `archway.ini` file to true to make it easier on yourself to determine what is going on in the changes you make.

1. Log in to the Get.It! Administration Module by logging into Get.It! with a user ID that has administration rights.
2. Click **Admin** to access the Get.It! Administration Module.
3. In the activities, click **Settings**.
4. In the section titled *General Execution Options*, enter a path and file name in the Log file field. This is where the debug information will be put.

5. Set the *Debug logging* option to **true**.
6. Set the *Debug Mode* option to **true**.
7. Click **Save** to save your changes (scroll down to below the settings table to find the Save button).
8. Be sure to set these options back to **false** before you release your changes to your entire user base.

You should also enter the following line into a script you are changing; it will output debug information to the archway log file as specified in the Get.It! Administration Module Settings. Whatever you type for String Value is what will appear in the `Archway.log` file.

```
env.debuglog( String Value);
```

Where to Make the Modifications

If you change the files we send with the Get.It! weblication, your changes will be lost the next time you install a new version of Get.It! To prevent this, a directory called “user” has been included within the `...\getit\apps\` directory.

Within the “user” directory are three directories called “schema,” “jscript,” and “weblication.”

- If you are updating a schema, save the updated file in the `schema` directory.
 - If you are updating a script, save it in the `jscript` directory.
 - If you are updating an XML file, save the updates in the `weblication` directory.
1. Open the file you want to change. This could be a schema file, an application file, or a script.
 2. Use the Save As command to save the file into the `...\getit\apps\user\` directory. All files saved in the correct subfolder (schema, jscript, weblication) in this directory are read by the `wbuild` command.
 3. Make your changes and save the file.
 4. Run `wbuild application`, where “application” is the name of the application you are changing.

You can test how this works using the following method:

1. Find the sample `prgn.xml` file in the `...\getit\apps\user\weblication\samples\` directory. You can use any of the files in this directory as a test, but in this example we will use the `prgn.xml` file.
2. Copy the `prgn.xml` file from the `...\samples\` directory into the `...\getit\apps\user\weblication\` directory. After running the `wbuild` command, log into Get.It! and you will see a new “PRGN” tab in the Get.It! header menu.

Printing Forms

You can include HTML forms that are designed with an external tool or editor, thus allowing you to print to a default printer. There is an example of how to implement this functionality in the `... \apps\user.xml` directory. The following illustrates how to implement this kind of form using a third-party editor.

This example uses the Health and Human Services Form 393. To enable this sample, edit the `... \apps\user.xml` file and remove the X in `<Ximport href="samples/hhs.xml" />`.

Get.It!

User: mmccool
Form: request
Activity: detail
Module: status

Home
Admin
Request
Request Status
Show My Requests
REQ000038 Status
Approvals
Receiving
Assets
Answers
Service
B2BAdmin

Status for request REQ000038: Pending approval

This request has been submitted and requires approval. See the approval workflow for details.

When would you like this and what is it for?
Date: 9/18/00 Purpose: Upgrade

Who is this for and where should it be delivered?
First: Matthew Location: -26612
Last: Mcceool Address:
Phone: City:
State:
Zip Code:

Which department is paying for this?
Cost Center: Comments:
Project:
Budget: Attachments: - Select a file -

Request contents: (select each item for order and delivery details)

Count	Brand	Model	Description
1	IBM	M PRO 8500 9.1GB 256MB 512K 6/6	IBM M PRO

Grand Total: \$2,629.00

Approval Workflow Go Back

Fig. 4.4 Get.It! Form Version

Prepare Report for Default Printer (browser directs the print spool).

1. Define a weblication form (e.g., `user\weblication\hhs.xml` as shown in the example below).

```
<!-- The following adds an activity to the Status module that
displays a request as a "393 Form" as defined by the HHS government
agency.-->
<module name="status">
<activities>
<activity name="hhs" menu="detail;workflow;hhs">
<description short="393 Format" long="View request as 393 form fit
for printing"/>
</forms>

<!-- This form displays the policies -->
<form name="start" sidebar="false"
onload="procure.loadRequestDetails">
<jspimport href="form393.htm"/>
</form>
</forms>
```

```

</activity>
</activities>
</module>

```

All of the bold text above requires customization. The activity's menu attribute includes the names detail, workflow, and hhs so that this activity appears in the Activity menu. Note that the `sidebar="false"` attribute is unnecessary for version 1.3 and above. At present, this is required only for backward compatibility with the older Module Tabs implementation where an activity menu is included within the frame with the form.

After defining the weblication form, run `wbuild` to create the new JSP page for this HHS activity.

2. Copy the externally created HTML file into the `...\user\presentation\` directory (e.g., `form393.htm` in our example). Browse through this HTML file and locate all required dynamic data. An example may appear as follows:

```
<td><span style='font-size:8.0pt'>REQUISITION NUMBER:</span></td>
```

The actual value for this field will be found in the `<Number>` element of the returned XML message from the executed onload JScript (e.g., `procure.loadRequestDetails`).

3. For each needed dynamic data value, add a JSP expression to insert the actual value into the proper place in the form. For example:

```
<td><span style='font-size:8.0pt'>REQUISITION NUMBER: <%= msg.get(
"Number", user ) %></span></td>
```

This sample JSP expression extracts the `Number` element from the XML document returned by the JScript routine `procure.loadRequestDetails`, as defined above in the weblication form.

The XML message returned from the JScript will look similar to the following:

```

<Request>
  <Id>26370</Id>
  <EndUserName>Hartke</EndUserName>
  <Number>REQ000040</Number>
  <RequestedFor>2000-8-9</RequestedFor>
  <TotalCost currency="USD">9573.00</TotalCost>
  <RequestLines _count="-1" _countFound="1" _more="0" _start="0">
    <RequestLine>
      <SupplierName>COMPUCOM</SupplierName>
      <Name>PL 6500R PIII XEON/500 512KB 256MB</Name>
      <Quantity>1</Quantity>
      <TotalPrice currency="USD">9573.00</TotalPrice>
      <DateNeeded>2000-8-9</DateNeeded>
    </RequestLine>
  </RequestLines>
  <EndUser>
    <Id>11078</Id>
    <EndUserPhone>(650) 572-9000</EndUserPhone>
  </EndUser>
  <ApprovalStatusName>Pending approval</ApprovalStatusName>

```

```
</Request>
```

4. Save the HTML file.
5. To test your work, navigate to the module that contains the new activity (e.g., 393 Format).
6. Right-click on the pop-up window and select Print.

Necessary Information

If you create your own file, or if you want to save just the part of the module you change in the `...getit\apps\user\` directory, there are four items you must have at the beginning and end of every file. (In the sample below, replace `xxx` with the name of the module and `yyy` with the name of the activity.)

```
<module name="xxx">
  <activities>
    <activity name="yyy">
      <forms>
        ...

      </forms>
    </activity>
  </activities>
</module>
```

Running the wbuild Command

The `wbuild` command, as explained earlier in this chapter, takes all of the ingredients and generates a set of web pages that are installed into a web server directory. You can enter parameters to have specific applications compiled.

`wbuild application =` Compiles the application you enter. Replace `application` with the name of the application you want to compile, such as `wbuild getit`. Run `wbuild getit` unless you have created your own application definition file. The valid application parameters are those set in the Application Definition file. See “Application Definition File” on page 4-4 for more information on this file.

For Solaris, since the file name is `getservice.xml`, you would run `wbuild.sh getservice`.

1. Display a command prompt. One method of doing this is to use `Start>Programs>Command Prompt`. Change the directory to `...\getit\bin`. (To change the directory, first ensure you are at the correct directory, for example a `C:>` by typing `C:` and pressing ENTER. You should see `C:\>` as your prompt. Then type `cd program files\getit\bin` and press ENTER. You should now see `C:>Program Files\getit\bin>` as your prompt.)

2. Type `wbuild appname` at the prompt and press ENTER. Where `appname` is the name of the application package you want to recompile. The `wbuild` command processes all weblication files in each package. It will list all the processing it is going through. When the processing is complete, you can minimize or close the window.

Changing Form Contents

Each form in a weblication is defined by a `<form>` element in its appropriate module file. This is where form contents are declared, including things such as Title, Instructions, Fields, Menus, Tables, Links, and Action Buttons.

Note: Make sure you store your changes in the `...getit\apps\user\` directory.

You can add to or delete from these contents. In the example below, we will add a field to a form.

Adding Form Fields

To add a field to a form, consider the following example. The sample below is taken from the Get.Resources! application, and it shows the details for a specific product in the company catalog.



Fig. 4.5 Adding a field to a form

Note: Make sure your form statistics are displayed. See “Displaying Form Information” on page 4-5 for instructions.

The form contains detail fields for the product, including the model, brand, list price, etc. The Form Statistics tells us exactly where to go for the form definition: we're viewing the product form in the catalog activity of the request module (and, therefore, found in `request.xml`).

In this example, we assume you have not previously modified the `request.xml` file. If you have, open the file from within your `...getit/apps/user/weblication` directory instead.

1. Open the `request.xml` file from the `...getit/apps/resources/weblication` directory.
2. Use the Save As command to save the file into your `...getit/apps/user/weblication` directory.
3. Find the form named `product` in the activity named `catalog`. The form is defined in the following manner:

```
<form name="product" onload="procure.getProduct">
  <title field="Model"> $$ (Model) </title>
  <fields>
    <field name="image" type="image" field="PhotoId"/>
    <break/>
    <field name="brand" label="Brand" field="Brand"/>
    <field name="description" label="Description"
      field="Description"/>
    <field name="price" label="List Price" field="Price"/>
    <field name="comments" label="Comments" field="Comment"/>
    <link name="infos" label="More Info" target-field="URL"
      window="true"/>
    <break/>
    <field name="vendor" label="Availability from Vendor"/>
  </fields>
  <table record="Supplier">
    <column label="Vendor Name" field="Name"/>
    <column label="Availability" field="Available"/>
    <column label="Delay" field="Delivery"/>
    <column label="Price" field="Price"/>
  </table>
  ...
  <actions target-form="additem">
    <submit> Add to shopping cart </submit>
    <back/>
  </actions>
</form>
```

Consider how to add a **Delivery** field to the form that displays the average time it takes for the catalog item to be available once ordered. This is achieved by adding a field entry to the form, as shown below. The revised XML below contains this new field:

```
<form name="product" onload="procure.getProduct">
  <title field="Model"> $$ (Model) </title>
  <fields>
    <field name="image" type="image" field="PhotoId"/>
    <break/>
    <field name="brand" label="Brand" field="Brand"/>
```

```

<field name="description" label="Description"
  field="Description"/>
<field name="price" label="List Price" field="Price"/>
<field name="Delivery" field="Delivery"/>
<field name="comments" label="Comments" field="Comment"/>
<link name="infos" label="More Info" target-field="URL"
  window="true"/>
<br/>
<field name="vendor" label="Availability from Vendor"/>
</fields>
<table record="Supplier">
  <column label="Vendor Name" field="Name"/>
  <column label="Availability" field="Available"/>
  <column label="Delay" field="Delivery"/>
  <column label="Price" field="Price"/>
</table>
...
<actions target-form="additem">
  <submit> Add to shopping cart </submit>
  <back/>
</actions>
</form>

```

4. Save the modified file in the `...getit\apps\user\weblication\` directory.
5. After making this modification, run the `wbuild getit` command to regenerate the form. See “Running the wbuild Command” on page 4-11 if you need instructions.

The modified window in the browser is displayed below.

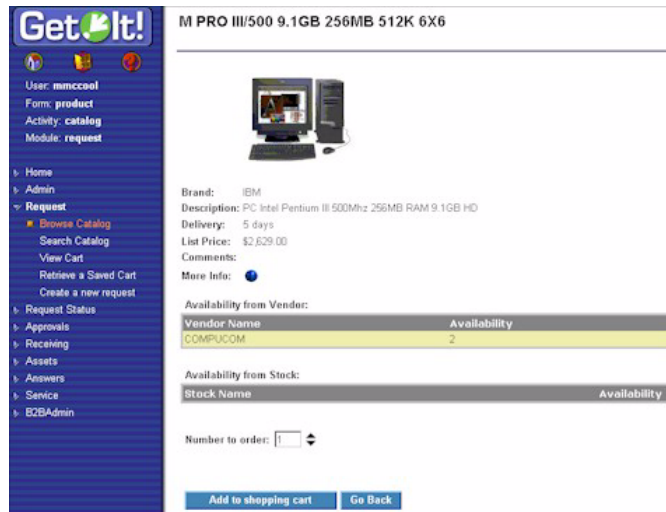


Fig. 4.6 With the new field.

Data for the New Field (Scripts)

A remaining question is where does the **Delivery** field actually come from? All data available to a script is provided by its `onload script`. This is defined in the form's declaration:

```
<form name="product" onload="procure.getProduct">
```

As shown, the form in our example relies on the `getProduct` function of the `procure` script file. This script is designed to return product documents. The product document schema includes the Delivery field we just added.

Of course, this tells us that the type of change described in this section is possible as long as the data for a new field is already provided by the form's script. This is not always the case. In order to display new fields, it is sometimes necessary to modify Document Schemas or even the script logic. The next two sections describe how to do this.

Split Frames (forms)

If you are working with forms you have the option of displaying two or more forms on a single page. This allows a master list of items to be displayed on top and the details of a selected item to be displayed at the bottom. The following example illustrates the use of `rows` to accomplish this. Modify the bold text to accommodate your frame dimension requirements.

```
<form name="start" type="frameset" rows="335,*">  
  <redirect name="lsit" target-form="results"/>  
  <redirect name="detail" target-activity="detail"/>  
</form>
```

Localizing Your Changes

If you want to translate the change you have made to other authorized languages, use the following steps. See Chapter 7, "Localization Support," for more information on localizing your changes.

1. Make sure you verify the change is working in the English version of the weblication and that `wbuild` has been run.

Open a command prompt (like you do when you run `wbuild`) and run `strbuild getit`. This will create the STR files for the languages you have set in the Administration Module Settings.

The translated *.str files are created only if the "resolve-AppStrings" field in the archway.ini file is set to True. If it is set to False, only the English string file is created.

2. If your allowed languages include more than English (based on the entries in the Locale field in the Administration Module Settings), you will need to update the appropriate string files with the translation of the new field you added. For this example, we will update the French file.
3. Open the `getit_fr.str` file and search for the field you just added. In this example, search for “Delivery.”

In this example, we are using Notepad to update the file. If you use a different program, your window may look different.

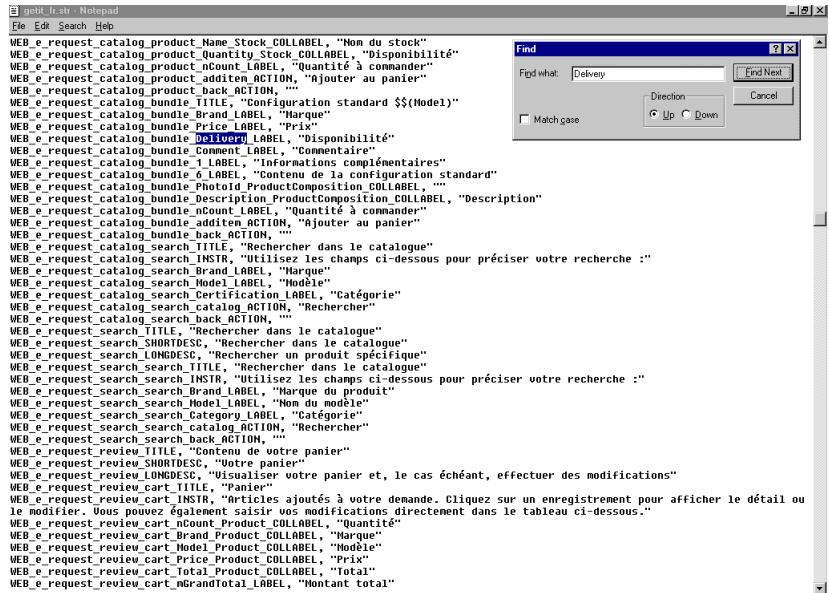


Fig. 4.7 Translating the new field.

4. The new field will show blank quotes unless this string has been previously translated into this language. Type the translation of the string into the blank quotes. In the example above, we have typed the translation for Delivery.
5. Save the file. Do *not* save it into a `user` directory.
6. Log into Get.It! and verify that the translation was successful.

Adding Fields to a Document

Most of the scripts in Peregrine Systems’ Weblications use Archway’s Document Manager to exchange data with back-end systems like ServiceCenter or AssetCenter. See “The Document Manager” on page 2-10 for an introduction on the Document Manager.

One of the main reasons for using the Document Manager is that it makes customization possible without the need to modify database operations hard coded in scripts. If your customization needs call for adding more data to a document, you can do it by extending the appropriate Document Schema.

Undefined Schema Fields

To add a field that is not yet defined in the schema, consider the following example. When a user enters a request in the Get.Resources! application, the following screen queries for various fields describing the request:

Get.It! Status for request REQ000038: Pending approval

This request has been submitted and requires approval. See the approval workflow for details.

When would you like this and what is it for?
 Date: 9/16/00 Purpose: Upgrade

Who is this for and where should it be delivered?
 First: Matthew Location: - 26612
 Last: McCool Address:
 Phone: City:
 State:
 Zip Code:

Which department is paying for this?
 Cost Center: Comments:
 Project:
 Budget: Attachments: - Select a file -

Request contents: (select each item for order and delivery details)

Count	Brand	Model	Description
1	IBM	M PRO III/500 9.1GB 256MB 512K 6/6	IBM M PRO

Grand Total: \$2,629.00

Approval Workflow Go Back

Fig. 4.8 Adding information to a schema

Note: Modifications of the schema are restricted to fields that exist in the database.

The form allows the user to specify a request purpose, delivery date, cost center, etc. Now let's assume that we want to add a new field to track the requester's Internal Credit Number—a company specific number given to each employee.

The new company-specific field obviously does not exist out-of-the-box in the document associated with this form. The document used by this particular form is the request document and the schema is the file that defines which fields are available. Each schema file contains a generic document definition, followed by one or more system-specific derivations. In other words, the first portion of the schema defines the fields for Get.It!, and the second portion of the schema maps the Get.It! field to the field in a table in one of the back-end systems. To begin to add a new field:

1. Open the *request.xml* file from the `...getit\apps\resources\schema\` directory.
2. Use the Save As command to save the file into your `...getit\apps\user\schema\` directory.
3. Find the “request” generic document definition, which is shown below:

In this example, we assume you have not previously modified the *request.xml* schema. If you have, open the file from within your `...getit\apps\user\schema\` directory instead.

```

<document name="Request">
  <attribute name="Id" type="num"/>
  <attribute name="ApprovalStatus" type="num"/>
  <attribute name="Budget" type="string"/>
  <attribute name="Comment" type="string"/>
  <attribute name="CostCenter" type="string"/>
  ...
</document>

```

4. To add our new Internal Credit Number, we start by inserting the field into the “request” generic document definition:

```

<document name="Request">
  <attribute name="Id" type="num"/>
  <attribute name="ApprovalStatus" type="num"/>
  <attribute name="Budget" type="string"/>
  <attribute name="Comment" type="string"/>
  <attribute name="CostCenter" type="string"/>
  <attribute name="ICN" type="num"/>
  ...
</document>

```

This new line defines a new numeric field named ICN. The field has been added to the generic request schema definition. This definition is generic because it is not tied to any specific back-end system. However, because the Get.Resources! application is implemented on top of AssetCenter, we also need to extend the AssetCenter specific request schema.

5. Each schema file contains a generic document definition followed by one or more system-specific derivations. You can see the AssetCenter schema for request in the `...getit\apps\resource\schema\request.xml` file. Here is the line added to that definition:

```

<documents name="ac">
  <!-- AC Request Document -->
  <document name="Request" table="amRequest">
    <attribute name="Id" field="lReqId"/>
    <attribute name="ApprovalStatus" field="seApprStatus"/>
    <attribute name="Budget" field="Budget.Name"
      link="lBudgId" linktable="amBudget" linkfield="Name"/>
    <attribute name="Comment" field="Comment.memComment"
      link="lCommentId" linktable="amComment"
      linkfield="memComment" linktype="hard"/>
    <attribute name="CostCenter" field="CostCenter.Title"
      link="lCostId" linktable="amCostCenter" linkfield="Title"/>
    <attribute name="ICN" field="Field2"/>
    ...
  </document>

```

</documents>

The purpose of entries in the AssetCenter specific schema is to define the mapping between a logical document field and its AssetCenter physical database counterpart. In this case, we've mapped the new ICN attribute to Field2 in the amRequest table. Field2 is a customizable generic field in the AC database, and in this example we have chosen to use it for storing the ICN number.

6. Save the changes you made to the `...getit\apps\user\schema\request.xml` file.
7. After making this modification, run `wbuild getit` to regenerate the form. See "Running the wbuild Command" on page 4-11 if you need instructions.

With just these two new lines in the request document schema, the Weblication is now capable of tracking a new field with every request. Now we can add the field to any form in the same way described in the previous section.

Changing Script Behavior

The Get.It! architecture is designed to minimize the need for script changes, however, you can customize the logic of an Archway script. The Document Manager minimizes the number of modifications you might make, because, as described in the last section, you can modify the type of data returned by a script by simply updating the appropriate Document Schema.

However, for those times when you must modify a script, the Archway's script model allows you to make modifications without having to alter the base code shipped by Peregrine Systems. You just create your own version of the function in a user-derived script. As with all other items you modify, store your user-derived scripts in a directory separate from the scripts shipped by Peregrine Systems. This directory is in `...getit\apps\user\jscript\`.

Changing JScript

Consider the following example. The following screenshot shows a form in the Resources module. The form is used to enter data describing a request.

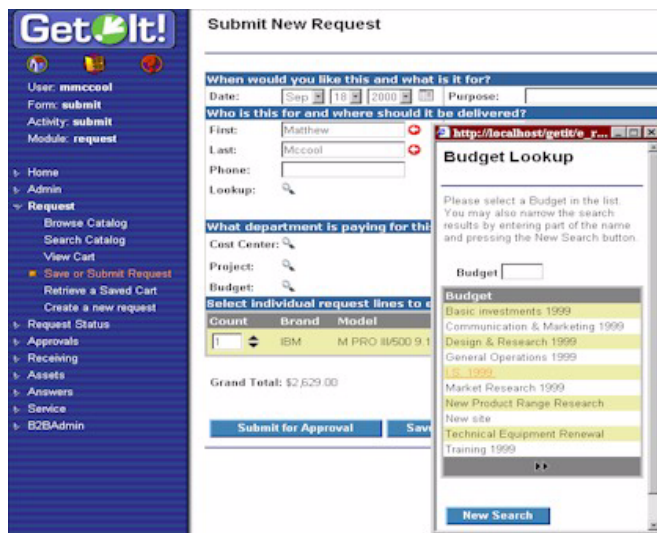


Fig. 4.9 Changing JScript.

This form includes selection boxes that are populated with valid choices obtained by queries against the database. For this example, we will add another field to this form to capture the requester's Department in the company. To accomplish this, we will need to modify the form's script to query for a list of valid department names that can be shown in a new select box.

1. Determine which script is used. You can do this by looking at the form's `onload` script, which is specified in the form's XML definition. Use the Form Statistics to determine where to look in the XML file. In figure 4.9 above, the form is defined in the `submit` activity of the `request` module.
2. Open the `...getit\apps\resources\weblication\request.xml` file.
3. Search for the form named `submit`, in the activity named `submit`. Here is the form's declaration:

```
<form name="submit" onload="procure.getOrderParameters">
```

4. Determine the name of the javascript file by looking at the onload element. The `getOrderParameters` function of the `procure` script is responsible for gathering data for the form. The contents of the script can be found in the `procure.js` script file.
5. Open the file called `procure.js` from the `...getit\apps\resources\jscript\` directory.
6. Save the file in the `...getit\apps\user\jscript\` directory.
7. Within this file, find the following code:

In this example, we assume you have not previously modified the `request.xml` file. If you have, open the file from within your `...getit/apps/user` directory instead.


```

function getOrderParameters( msg )
{
    ...
    // Get the list of Budgets
    msg = new Message();
    msg.add( "_return", "Name" );
    msg.add( "_sort", "Name" );
    msg = archway.sendDocQuery( "ac", "Budget", msg );
    msgResponse.add( msg );
    ...
}

```

8. Now you need to extend the work of the default script to include a new query for company department names. The following is the new user function in its entirety and then consider each of its lines of code:

```

function getOrderParameters( msg )
{
    var msgResult;
    var msgDepartments;

    // Call base function to perform standard queries
    msgResult = this.parent.getOrderParameters( msg );

    // Query for departments
    msgDepartments = archway.sendDocQuery(
        "ac", "SELECT Name from amEmplDept WHERE bDepartment=1" );

    // Add departments to overall response
    msgResult.add( msgDepartments );

    return msgResult;
}

```

9. Save your changes.

This procedure defines a new function with the same name as the one we're trying to extend (`getOrderParameters`). The new function is stored in a new user script file with the same name as the base script file (`procure.js`). By doing this, we're guaranteed that Archway will invoke our new function instead of the base version.

Within the function, we included a call to the base function:

```
msgResult = this.parent.getOrderParameters( msg );
```

It is not mandatory to do this. However, by calling the parent function, we preserve the base queries and only add our new query on top. In some cases, you will want to bypass the original behavior altogether.

Next, we query for the data of interest:

```
msgDepartments = archway.sendQuery(
    "ac", "SELECT Name from amEmplDept WHERE bDepartment=1" );
```

This gives us a result set with a list of department names. Finally, the list is added to the result set obtained from the base function:

```
msgResult.add( msgDepartments );
```

The only remaining task is to add the actual department field to the weblication form. We already saw how to do this in an earlier section.

Changing Weblication Components Layout (XSL)

The layout and organization of each form is determined by a set of template files. The templates are defined in the Extensible Stylesheet Language (XSL).

The purpose of XSL is to process an XML document and convert it into a different desired format. For instance, an XSL template could define rules for converting an XML document into HTML that can be displayed by a browser. A different XSL document could generate an RTF-like document that is more suitable for printing.

The XSL templates provided with the product are used by the `wbuild` command in conjunction with the weblication XML definition to generate web pages. Get.It! includes a set of templates that generate Java Server Pages (JSP) files.

When Do I Change the XSL?

There are two reasons for extending or customizing the templates provided by Peregrine Systems.

- To add support for a new type of weblication component
- To change the layout or organization of a web site

You can make the modifications without altering the existing template source files. Again, this is important for upgrade purposes. The source for XSL templates can be found in the `...\getit\templates\jsp` directory. The directory also contains a file named `user.xml`. This is where you can enter your own customization.

The `user.xml` file shows basic instructions and examples for customization. Template customization with XSL is an advanced topic and further description is beyond the scope of this guide.

Integrating a New Product with Get.It!

The method you use to integrate new products into Get.It! depends on the type of product you want to integrate.

Integrating a URL

If you are linking a product that can be accessed through a web browser using a URL, you can add the product as a new module or as an activity on an existing module.



Fig. 4.10 Adding a URL as a module or as an activity

Note: If you integrate a URL as an activity, it will be available to users as part of the activity list for an existing module.

Adding a URL as a Module

The default path is the path we recommended at installation. If you installed Get.It! into a different folder, your default path will be the path you chose at installation.

If you add the URL as a module, users can access it through a button on the main menu and on the menu bar.

1. Create a new XML file.
2. Save the new file in the `...\getit\apps\user\weblication` directory.
3. Open a file from one of the Get.It! application packages that most closely matches the module you want to add. There are also samples in the `...\getit\apps\user\weblication\samples` directory.
4. Copy the tags from the existing file and paste them into the new XML file you created in step 1.
5. Update the module name with the name of the product you are integrating. Make sure to change the module name, title, image, short description, and long description to match the module you are adding. Update the “target-url” with the URL of the product you are integrating.

6. Save the new file in the `...\getit\apps\user\weblication` directory. Make sure it has an extension of XML.
7. Run `wbuild getit`. See “Running the wbuild Command” on page 4-11 if you need detailed instructions.
8. Log out and log back in to Get.It! and the new module is available.

Adding a URL as an Activity

If you add a URL as an activity, users can access it through a link on the activity list in an existing module.

1. Log into Get.It! and determine the module in which you want the new activity to be available.
2. Determine if there is an activity that behaves similarly to the activity you are adding. For example, is there an existing activity that links to a different URL?
3. Open the XML file from an existing application package (in the `...\getit\apps\` directory) for the module into which you want to integrate the new activity.
4. Use the Save As command to save this file into the `...\getit\apps\user\weblication\` directory.
5. Find the section of the XML file where the activities are defined and enter the following, replacing the Peregrine Systems information with the information for the URL you want to integrate.

```

<!--=====
      Activity: link to Peregrine
=====-->

<activity name="prgn">
  <description
    short="Peregrine"
    long="Link to Peregrine's web site."
    target-url="http://www.peregrine.com"
  </activity>

```

6. Update the name of the activity to be the name of the product you are integrating. Make sure to change the activity name, short description, and long description to match the product you are integrating. Update the “target-url” with the URL of the product you are integrating.
7. Save your changes.
8. Run `wbuild`. See “Running the `wbuild` Command” on page 4-11 if you need detailed instructions.
9. Log out and log back in to Get.It! and the new activity is available.

Adding a New Module

Adding a new module requires you to copy an existing XML file, make your updates, and then save the new module to the `...\getit\apps\user\weblication\` directory. You define the module in the new XML file and then add it to the Weblication when you import it.

To start, determine an existing XML file that is the closest to the new module.

1. Open an XML file from an existing application package (in the `...\getit\apps\` directory) that you want to change, or that does a similar action to what you want the new module to do. If no existing XML closely matches what you want to do, we recommend you still open a file to use as a guide.
2. Use the Save As command to save this file in the `...\getit\apps\user\weblication\` directory with a name that allows you to easily recognize what this module does. Remember to include the XML extension on the file.
3. Update the applicable portions of the file, including header information, nested tags, etc. Update the new XML file until it includes all the functions that you want it to do. Use the instructions in the previous sections of this chapter.

4. If you need to populate tables in the new module, you may need to create a new script in the `...\getit\apps\user\jscript\` directory. Copy an existing script, just as you did to create the new XML file. When you save the new script, be sure to include the JS extension on the file name. See “Changing JScript” on page 4-19 for instructions on updating a script file.
5. Run `wbuild getit`. See “Running the wbuild Command” on page 4-11 if you need detailed instructions.
6. Log out and back into Get.It! to see the changes you have made.

Modules can be removed from a Weblication by removing their entry in the `...\getit\apps\user\` directories (if they are modules you created) or from the `getit.xml` file if they are modules that came with Get.It!.

Adding a Feature from AssetCenter

Within AssetCenter, features may be added to track information not provided for by the out of the box database schemas. The Get.It! weblication allows features to be incorporated as well, allowing customization of the databases and screens for use by all users.

1. Add the feature to the desired table within AssetCenter. This should be done in the typical AssetCenter fashion.
2. Add access to the feature via `amUserRight` entry. You must give access to the feature via `amUserRight` modification. Select the `amUserRight` entries for which the new feature is relevant and provide access as necessary.
3. Add the feature to a schema. Once the feature has been created within AssetCenter, add it to the weblication’s schema. An excerpt from the `request.xml` schema is shown here. The necessary addition has been highlighted in bold. See “Adding Fields to a Document” on page 4-16 for details on updating a schema.

```
<schema>

<documents name="base">
  <!-- Request Document -->
  <document name="Request">
    <attribute name="Id" type="num" />
    [...]

    <attribute name="TestFeature" type="string" />
    [...]

  </document>
</documents>

<!--=====
```

AssetCenter Schema Derivations

=====-->

```
<documents name="ac">

  <!-- AC Request Document -->
  <document name="Request"          table="amRequest">
    <attribute name="Id"            field="lReqId"/>
  [...]

    <attribute name="TestFeature"   field="fv_TestReq"/>
  </document>
</documents>
```

4. Add the feature to an application. After the feature is referenced in the schema, you need to incorporate it in to the screen definitions. See “Changing Form Contents” on page 4-12 for details on updating an application.

5. An example is given here from the `...getit\apps\request.xml`:

```
<!-- This form requests order information for submission -->
<form name="submit" onload="procure.getOrderParameters">
  <title> Request Information </title>
  <instructions>
    Please provide the following information necessary
    for submitting your request.
  </instructions>
  <fields>
    <input label="Purpose" type="text" field="Purpose"
      size="50"/>
  [...]

    <input label="Test Feature" type="text"
      field="TestFeature" />
  </fields>
```

6. Run `wbuild getit`. See “Running the wbuild Command” on page 4-11 if you need detailed instructions.

Portal

The Portal is the Get.It! interface, customizable to your specific requirements. There are numerous Components that comprise the Portal. Each Component has a different function, some of which are also customizable. You can modify the layout, open and close Components, or remove Components from the interface. The Portal that you receive upon installing Get.It! will look similar to the following. This is an example of one Component that you have access to within Portal. There are many more Components, each divided

amongst two categories. The first category is Weblications, including Application Menu, Problem Tickets, Request To Approve, and Status Review. The second category is Personal Utilities, including Calendar, Calculator, Weather, Stock Quotes, Top News, Business News, Technology, Sports News, and Date And Time.

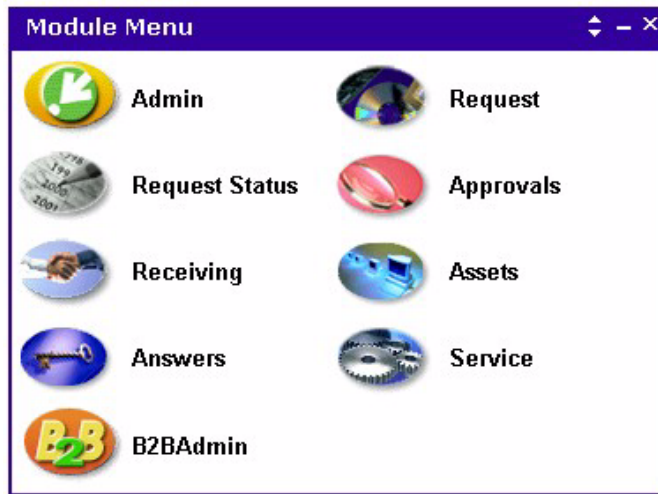


Fig. 4.11 Module Menu Component

Note: If you are using Netscape Navigator as your default browser, some screens may appear differently than stated. Notes are given where differences occur.

Customizing (GUI)

The Get.It! Portal is a dynamic interface. You may move and remove all Components. The Stock Quotes Component is modifiable, adjustable to those stock quotes that you specify. Each Component contains a toolbar in its upper-right corner. Depending on the type of Component, each corner will contain Customize (wrench), Copy (superimposed windows), Move (vertical arrows), Hide (dash), and Remove (X) options.

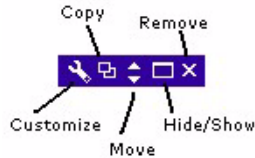


Fig. 4.12 Window Tools

Customize

The Customize option is for customizing the Component. Not all Components are customizable (e.g., News, Sports). Clicking on the Customize button will link you to the Change Layout screen, where you can modify the layout. To save your modifications click on the Save button at the bottom of the screen.

Copy

Copy allows you to copy a Component. The only Component that you can copy is the Problem Tickets Component. When you copy the Problem Tickets Component, the second Component will be identified by its title within the title bar (e.g., Problem Tickets: Copy 1).

Move

The Move arrows move Components up and down. You cannot move a Component left and right. To move a Component up press the up arrow. Pressing an arrow will move the Component by one increment. For instance, if the Weather Component is on top and the Stock Quotes is directly beneath, pressing the down arrow on the Weather Component will move it one notch down, or below the Stock Quotes Component.

Hide/Show

Hide closes the Component, reducing the window to a menu bar with selectable options (i.e., Window Tools). If a Component is in a Hidden state, then the option becomes Show. Pressing the Show button will display the Component. The Hide and Show buttons alternate.

Remove

Pressing the Remove button removes the selected Component from the portal. When you press Remove, a prompt will appear: Remove This Component? Click OK to remove the Component or Cancel to void the process. You can always reinstate a Component once it is removed by pressing the Select Content link and then selecting the desired Component.

Select Content

Clicking on Select Contents links you to Customize My Home Page. Within this page are two main features, each containing multiple options that allow you to customize your home page (or Portal). Each feature allows various selections to be made. You can customize anything from Problem Tickets and Status Reviews to personal Stock Quotes and the Weather. All of these features are available as Components on your front page.

You do not have the option of customizing all Components. For instance, if you select the Weather option located within Personal Utilities, you will not have the option of locating the forecast for your specific area. Selecting the Weather option provides a United States map with various forecasts from around the country. There are other Components that are customizable. For instance, the Stock Quotes option allows you to edit the displayed stock (the default for this option is PRGN). All of the news options are provided by iSyndicate and are not customizable.

Get.It! Weblication Components

Within the Get.It! Weblication Components, you can select Application Menu, Problem Tickets, Requests To Approve, and Status Review. Each of these options is selectable by clicking on its check box. A check mark in the box means that the option is selected and that it will appear on your home page. Each selection is accompanied by a brief description located to its right. Once a selection is made it must be saved. Saving is accomplished by clicking on the Save button located at the bottom of the Select Contents screen.

Component	Function
Application Menu	Quick links to various modules that comprise the application.
Problem Tickets	Lists unresolved tickets. Click on a ticket to verify its status or to provide updated information.
Requests To Approve	Allows you to select the approval action for resource requests. You may click on any request to view more details. This component requires access to Peregrine Systems' AssetCenter.
Status Review	Displays the list of active requests so that you may view their status. This component requires access to Peregrine Systems' AssetCenter.

Personal Utilities

The Personal Utilities section allows you to customize your home page with a variety of up-to-date options. You can select items from the latest Business News to the Date And Time. There are numerous possibilities. Select each option (check box) that you want for the home page and press the Save button located at the bottom of the page.

Component	Function
Calendar	A monthly calendar. This component can be viewed using MS Internet Explorer 4.0 or higher.
Calculator	A standard calculator. This component can be viewed using MS Internet Explorer 4.0 or higher.
Weather	A map of the United States containing various forecasts around the country.\
Stock Quotes	The latest quotes of your most frequented stocks.
Top News	The latest headlines provided by iSyndicate.
Business News	All things business provided by iSyndicate.
Technology	All things technological provided by iSyndicate.
Sports News	All things sport provided by iSyndicate.
Date and Time	Your choice of Coordinated Universal Time (UTC or GMT) or Standard Local Time.

Create New (web page)

Create New, located at the bottom of the Select Content page, allows the creation of a new web page. To create a new page you must insert a Title, a Display URL Link, and a Configure URL Link. The Title is the title of the page. This is synonymous with the HTML `<Title>` tag. The Display URL (Uniform Resource Locator) Link is the address where the page will be located. The Configure URL Link is where the page will be configured. There are also two options for page layout: Wide and Narrow.

For instance, in the Display URL Link you can insert `http://<your-host>getit/timedate.asp` if you are running IIS or JSP. For the Configure URL Link you can insert `http://<your-host>/getit/timedate_configure.asp` or JSP. The Date And Time component uses the JSP version. The ASP version is included as an example of how to incorporate Active Server Page technology within Get.It!

Title:	<input type="text"/>
Display URL Link:	<input type="text" value="http://"/>
Configure URL Link:	<input type="text" value="http://"/>
	<input type="radio"/> Wide <input checked="" type="radio"/> Narrow

Fig. 4.13 Create A New Page

Create A New Page

1. Click on Select Content.
2. Click Create New (at the bottom of the page).
3. Insert a Title.
4. Insert a Display URL Link.
5. Insert a Configure URL Link.
6. Click Save.

Change Layout

Your portal or home page is divided into two areas: Narrow for the left and Wide for the right.

The Narrow section occupies about one-third of the screen; the Wide section occupies about two-thirds of the screen. These proportions are not modifiable. Within both the Narrow Components and Wide Components sections are Up, Down, and Remove buttons. These buttons either move or delete a component. All active components are listed within the drop-down box.

To move a component up or down, or to remove it, first select the component in the drop-down box and use the arrow and delete keys to its right. Press the Save button at the bottom of the screen to save all changes of your layout. You may modify the layout at any time. While this functionality exists within the portal or main page, the Change Layout option allows you to customize the look more efficiently.

Narrow Components (shown on the left)



Wide Components (show on the right)

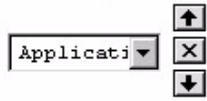


Fig. 4.14 Change Layout

Netscape Navigator

If you are using Netscape Navigator as your default browser, the Change Layout link allows you to move components either up or down. This feature is the same as the Component Move button (for Internet Explorer users).

Edit Preferences

The Problem Tickets component, part of the Get.It! Weblication Components section, provides an editing option for customization. Each field allows you to select information that is relevant to this component. You can select fields for: Status, Priority, who it is Assigned To, who it is Opened By, and Contact. You can also Add, Move, or Remove Field Choices, those items that are displayed in your table. You can also delimit the number of tickets to show. This illustration has 25 in its text box. This means that as many as 25 Problem Tickets can display at any given time.

Status: Open

Priority: Medium

Assigned To: Work Manager

Opened By: Robert Smith

Contact: User

Field Choices **Show in my table**

ID
Owner
Open Time
Status
Assigned To

Add

↑
X
↓

Max number of tickets to show: 25

Fig. 4.15 Customize

Portal API

Portal is customizable. You can add, remove, and modify Components. Portal gives you the opportunity to use HTML plug-ins or Active X components.

Located within the following directory are five samples of code, each containing highlighted script (in bold) that should be modified when making it user-specific.

```
... \getit\src\apps\user\weblication\samples
```

These samples include (1) Adding A New Module, (2) Overriding A Form, (3) Adding A New Activity, (4) Adding A New Activity Link, and (5) Changing the Requestform Component. Each of these samples is explained, detailing those sections of the code that must be modified when you initiate changes.

Sample	Function
Add Module	Adds a new module menu entry associated with another web site.
Override Form	Changes the login form to remove the Register option.
Add Activity	Adds a new activity to the Service module to allow searching for an arbitrary ticket.
Add Activity Link	Adds a new activity to the Resources module to display company request policies.
Modify Requestform Component	Changes the generic Get.Resources! Request Form by renaming several fields. This form is normally defined as a <code><component></code> entry in <code>components\reqform.xml</code> .

Portal Components

Similar to the `<component>` tag, the `<portal-components>` tag defines the individual windows found on the home page. These windows can be moved, minimized, and configured to provide the user with a concise view of multiple aspects of the Get.It environment. Portal components can also be linked to third-party plug-ins like news and weather, so that all the resources of the Internet can be brought together.

Out-of-box Portal components are defined in a file called `plugins.xml`, located in the `apps\portal\weblication\plugins.xml` directory. This file is structured as follows:

```
<portal-components>
<portal-category name="getit">
<title>Get.It! Weblication Components</title>
<portal-component
name="tickets"
copy="true"
adapter="SC"
access="getit.service">
<title>Problem Tickets</title>
<instructions>
Here is the list of unresolved tickets that you have reported. Please
click on any one of them to check the latest status or to provide us
with updated information. This component requires access to
Peregrine's <i>ServiceCenter</i>.
</instructions>
<contents onload="portal.getTickets">
<table record="Problem" columns="_Columns">
<link target-module="service"
target-activity="status"
target-form="detail"
field="Id" />
```

```

</table>
</contents>
<configure onload="portal.editTickets">
<fields>
<input type="select"
label="Status"
field="Status"
record="status"
valuelist="StatusValue"
displaylist="StatusDisplay" />
...
</fields>
</configure>
</portal-component>
...
</portal-category>
...
</portal-components>

```

<portal-category>

Attribute	Description
name	Defines part of the name used in the construction of the JSP file name.
<title>	Portal components are divided into categories. These categories define the caption that appears in the “Select Content” form that you can use for selecting the Components that you want on your home page.

<portal-component>

The portal component defines the content and configuration forms that are linked together.

Attribute	Description
<code>name</code>	Defines part of the name used in the construction of the JSP file name.
<code>copy</code>	If <code>copy="true"</code> , then an icon appears on the component toolbar that allows the configuration of a component to be duplicated. Use this in conjunction with <code><configure></code> to create multiple copies of a component configured to display different information.
<code>adapter</code>	Defines the adapter that is required to support this component. If that adapter is not accessible, then the component will not be listed in the "Select Content" form.
<code>browser</code>	Defines the browser that is required to display this component. If a user uses a different browser, the component will not be listed in the "Select Content" form.
<code>access</code>	Defines the access right that the user must possess to access this component. If a user does not have this access right, the component will not be listed in the "Select Content" form.
<code><title></code>	Defines the caption that appears on the toolbar of the component and next to the checkbox of the "Select Content" form.
<code><instructions></code>	Defines the text that describes the purpose of the component in the "Select Content" form.
<code><contents></code>	The contents of this element can be any weblication element, like <code><table></code> or <code><fieldtable></code> . This tag is equivalent to the <code><form></code> tag, including the ability to specify an onload script to gather data for the elements contained within.
<code><configure></code>	This optional form defines configuration parameters to be used by the <code><contents></code> form. If this tag is present, a wrench icon appears in the portal components toolbar. This tag is equivalent to the <code><form></code> tag. The result of this form submit is stored as <code><component.attributes></code> that are passed in the <code>msg</code> parameter of the <code><contents></code> onload function.

<plugin>

A plug-in element provides the user with the ability to reference third-party URLs to be included in a portal component. The URL can point to an HTML, JSP, or ASP page, or a GIF or JPG file from anywhere on the Internet.

Attribute	Description
<code>href</code>	Defines the URL address of the resource.
<code>post</code>	If the <code>post</code> attribute is set to “true”, then the data stored in <code><component.attributes></code> (see <code><configure></code> above) will be passed as post data to the URL.
<code>ID</code>	If the optional <code>ID</code> attribute is defined, the HTML response received from the URL will be scanned, and the HTML element with the given <code>ID</code> will be extracted from the result, discarding all other page information.

Portal Plug-Ins

Portal allows the use of plug-ins, small software programs that plug into a larger application to provide additional functionality. Plug-ins permit the browser to access and execute files embedded in HTML documents that are in formats the browser would not recognize such as animation, video, and audio files. Most plug-ins are proprietary.

The following XML script defines reusable plug-in components such as Stock Quotes. The complete version of this file can be found in `plugin.xml`.

```

<!--Generic Component Samples-->
<portal-components>
  <portal-category name="getit">
    <title>Get.It! Weblication Components</title>

  <!--AppMenu-->
  <portal-component name="appmenu">
    <title>Application Menu</title>
    <instructions>Links to various modules that comprise this
application.</instructions>
    <contents>
      <appmenu/>
    </contents>
  </portal-component>

  <!--Custom Component Container-->
  <portal-component name="custom" display="hidden">
    <contents>
      <custom-portal-component/>
    </contents>
    <configure>
      <custom-portal-configure/>
    </configure>

```

```

</portal-component>

<!--Yahoo Stock Quotes-->
<portal-component name="stockquotes" column="narrow">
  <title>Stock Quotes</title>
  <instructions>Monitor your personal portfolio.</instructions>
  <contents onload="portal.getStockQuotes">
    <table record="MyPortfolio">
      <column label="Symbol" field="Symbol"/>
      <column label="Price" field="Price"/>
      <column label="Change" field="Change"/>
      <column label="Link" field="Link"/>
      <link target-url="redirect.jsp" field="Link"/>
    </table>
  </contents>
  <configure onload="portal.editStockQuotes">
    <list-builder field="Symbol" size="5" default="PRGN">
      <src type="edit" label="Symbol" record="SymbolChoices"
displaylist="Symbol" valuelist="Symbol"/>
      <dest label="My Portfolio" record="MyPortfolio"
displaylist="Symbol" valuelist="Symbol"/>
    </list-builder>
  </configure>
</portal-configure>

```

Skins and Stylesheets

New skins and stylesheets can be selected from either the Skins & Stylesheets Portal Component or by using the navigational tree menu. There are several predefined themes, each consisting of a skin (images and graphics) and a stylesheet (fonts, colors, and special background images). You can select a new Skin, Stylesheet, or Both from the drop-down selection box located on the Portal home page or through the Skins and Stylesheets link.

New stylesheets can be created using the `Userschemes.xml` file and the `cssbuild` command. Working within the `Userschemes.xml` file, you must copy an entire `<sceme>` definition and then paste it in the same file. Rename the new `<scheme>` by changing the `<name>...</name>` element. Do not modify the names or locations of images because this will break the skin.

Creating New Stylesheets

1. Right-click on the `Userschemes.xml` file, located within the `... \program files \getit \templates \css \` directory, and deselect the Read-Only attribute.

2. Open `Userschemes.xml`.
3. Copy the first `<scheme>...</scheme>` (this is named “blues”).
4. Paste the new `<scheme>` definition in the same file.
5. Rename the new `<scheme>` (between the `<name>...</name>` element).
6. Modify applicable font colors and images (do not modify any of the elements).
7. Save the file.
8. Open a Command Prompt and run `cssbuild`.

Note: You must restart JRun to see the new stylesheet in the Stylesheets drop-down box.

All Skin and Stylesheet selections are predefined, thereby allowing you to quickly create a new portal design without modifying CSS, XML, or HTM files. It is important to understand the file structure when working with Skins and Stylesheets. The purpose of the directory structure is to provide image organization and user-friendly customization. You do not need to hard-code paths. All paths are dynamic and filled at runtime, allowing for instant updates without recompiling. You must use JSP files to process image paths. Other file formats will not be parsed by the JavaServlet engine. Do not modify the `Css.xml` file. If you add any new CSS files or Skins then place them in the `... \src\apps\user\presentation\` directory as a backup.

All image paths in the generated JSP are similar to the following.

Skinned Images

```

```

Non-Skinned Images

```
 and 
```

The paths are set by `Archway.ini` as in the following.

```
logo=images/logos/logo_getit_green.gif
images=images
skins=images/skins/
defaultskin=classic
```

When adding new images you will need to decide how it fits within the application structure. But remember that you will not create a fully qualified path for the image. In most instances the XSL file will suffice for the first half of the path.

All images are referenced in the XSL using token replacement, as in the following.

```
$$IMG(buttons/your_button.gif)
$$IMG(icons/your_button.gif)
$$IMG(backgrounds/your_button.gif)
```

There are other versions of this use in the XSL, as in the following.

```
$$IMG(<xsl:variable name="your var"/>)
```

\$\$IMG(@image)

Directory Structure

The following directory structure applies to all images, logos, and skins. All directories are dynamic, generated at runtime.

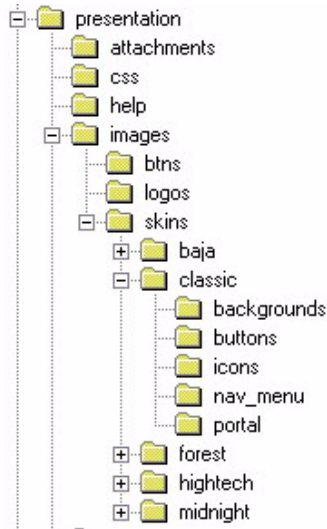


Fig. 4.16 Skins/Stylesheets Directory Structure

Directory	Purpose
Images	Generally, do not place images here. Use this directory only for images not set by skins; in other words, images that will never change and are common to all skins (e.g., <code>spacer.gif</code> , a transparent GIF for forcing spaces in table cells or forced breaks).
Logo	This is for application logos such as the Get.It! logo. These are not part of skins so they will never be affected in a skin or stylesheet change.
Skins	No images should be placed in this level. This file is for the organization of lower level skins directories.
Classic	The default skin. This directory can be copied, renamed, and placed within the <code>skins</code> directory. You can also place images at this level that do not fit elsewhere. You can also create new directories to further categorize images at this level.

Directory	Purpose
Backgrounds	All background images are placed in this file.
Banners	Currently empty, but suggested for organizing large banner-like images.
Photos	Suggested for any images used in applications.
Buttons	Defined as any graphic that is used as a button in the browser, except for icons.
Icons	Defined as the main button-like images used as the navigational elements in the application menu (i.e., oval-shaped image: <code>...graphics\logo\icon</code>).
Nav_menu	Used for all navigational sidebar or menu true images.
Portal	Used for all portal and portal component-related graphics.

XML Usage

```
<module name="b2badmin" access="getit.b2badmin">
  <title> B2BAdmin </title>
  <description
    image="icons/order.gif"
    short="B2BAdmin"
    long="Manage B2B sites and catalog data" />
  <columns>
    <column>
      <fields>

        <link target-url="e_b2bshop_return_b2blist.jsp"
          param="ListAction=CatalogUpdate" image="icons/capturecatitems.gif">
          Capture Catalog Items </link>

        <link target-form="ConfigureSites" image="icons/configsites.gif">
          Configure Sites </link>

        <link target-form="CheckVersions" image="icons/updateobjects.gif">
          Update Objects </link>

        <link target-form="BillingOptions"
          image="icons/billingoptions.gif"> Billing Settings </link>

      </fields>
    </column>
  </columns>
</module>
```

XSL (example)

Global_vars.xsl contains all the path variables for XSL documents.

```

```

JSP (example)

This is the minimum requirement in any JSP to parse the dynamic paths.

```
<%@ page language="java" buffer="16kb"
import="com.peregrine.archway.*, com.peregrine.archway.jsp.*,
com.peregrine.util.*, java.util.*,
java.beans.*, java.io.*, java.net.*"%>
<%@ include file="globals.jsp"%>

```

JAVA Source Code (example)

```
out.println( "<IMG SRC='" + Archway.getIni().getProperty( "skins",
"images/skins/" ) + Archway.getIni().getProperty( "defaultskin",
"classic" ) + "/buttons/calendar.gif' BORDER=0/></A>" );
}
```

External JavaScripts

This is more difficult in that any images written to the browser will not be affected by the skins. This is because JSP expressions for establishing image paths cannot be parsed. This can be resolved by creating variables in a JSP that is referenced by external JavaScript. The `Login.jsp` file is a good example. Make sure your `vars` are loaded before the external JavaScript file.

Note that all JavaScript `vars` are hard-coded into the XSL and then passed onto the `Login.jsp`. `Login.jsp` requires the external `Menubuilder.js` file because global JavaScript `vars` are established ahead of the `Menubuilder.js` call. They are then used by the JS file and parsed by the JSP to provide all necessary paths.

```
<script language="JavaScript">
var m_expandedIcon = "<img
src='<%=skins%><%=defaultskin%>/nav_menu/expand.gif' width='12'
height='14' border='0' />";
var m_collapsedIcon = "<img
src='<%=skins%><%=defaultskin%>/nav_menu/collapse.gif' width='12'
height='14' border='0' />";
var m_selectedIcon = "<img
src='<%=skins%><%=defaultskin%>/nav_menu/select.gif' width='12'
height='14' border='0' />";
var m_noselectedIcon = "<img
src='<%=skins%><%=defaultskin%>/nav_menu/noselect.gif' width='12'
height='14' border='0' />";
var m_blankIcon = "<img
src='<%=skins%><%=defaultskin%>/nav_menu/blank.gif' width='12'
height='14' border='0' />";
</script>

<script language="JavaScript1.2" SRC="js/menubuilder.js">
</script>
```

Production Environment

Before you enter the production environment you should note a few settings to help your transition.

```
java.args in(jrun n/jsm-default/properties/jsm.properties)
```


Be sure to set this parameter to something like the following.

```
java.args=-Xmx128m
```

This defines the size of heap memory given to JRun's Java VirtualMachin, providing 128 MB of memory. Larger settings may be appropriate on larger systems.

```
debugscript (archway.ini)
```

This should be set to false. (Starting with release 1.3, this is no longer important). Failing to do so will cause slower execution and substantial memory consumption.

```
scripttimeout (archway.ini)
```

This specifies the number of seconds allotted to each user to execute a script. The default setting is 30 seconds. If a user script runs longer than this maximum timeout value then all script operations are aborted.

Proper setting of this option is important to prevent run-away scripts from stopping all servers.

```
logstdout (archway.ini)
```

This should be set to false to avoid duplicate logging sent to stdout, potentially consuming disk space.

```
sessiontimeout (archway.ini)
```

Specifies the number of milliseconds to allow inactive sessions to exist before they expire. The default setting is as follows.

```
sessiontimeout=600000
```


This gives users 10 minutes before expiration. Sometimes smaller time limits may be appropriate. This setting is important because it ensures proper cleanup of old sessions that may potentially be holding on to required memory resources.

```
maxscriptrunners (archway.ini)
```

This setting controls the size of the FESI ScriptRunner pool created for Archway. A proper setting should rarely be greater than 10 (the default). However, some 1.2 systems may have shipped with a setting of 100. This setting is too large. Each ScriptRunner can consume a significant amount of memory, yet the efficiency of the system does not increase with a number of script runners greater than 10.

```
maxquerycount(archway.ini)
```

This is a new parameter added in 1.3 (originating with the 1.2 SP1 build update). If this parameter is set, it can limit the number of records that Archway will fetch for any given query. It is a good idea to set this parameter as a safety measure against runaway queries or bad scripts that attempt to fetch thousands of records into a single XML response. In some instances a system will need to process a large number of records. But the recommendation in such scenarios is to perform several queries for smaller record chunks. The Start and Count parameters to sendDocQuery() help accomplish this.



Chapter 5

Tailoring Get.Answers!

To customize Get.Answers! for IR Expert see your IR Expert documentation.

This section describes how to customize certain areas of Get.Answers! when you link to document collections and knowledgebases. You can change the following items for Get.Answers!:

- Change the color of the hit key words
- Change the title name of a knowledgebase folder in an advanced search
- Hide or show a hit list column
- Include a path name with the title of the document in the hit list

These items are specific to Get.Answers!. To change other areas of the weblication specific to Get.It!, see “Tailoring Get.It!” on page 4-1.

Modifying the Site Definition File

You can make the following changes to the site definition file in a text or XML editor:

- Change the color of the hit key words
- Change the title of a knowledgebase folder in an advanced search

To modify the site definition file:

1. Open the **GetAnswers.sdf** file in Notepad. It is located in the bin directory.

```
<?xml version="1.0"?>
<!DOCTYPE site SYSTEM "file:SiteDefinitionFile.dtd">
<site
    content-type="text/html"
    document="main-h.htm"
    hidden="no"
    highlight-style="font-weight: bold; color: #ffcc00;"
    id="1.3/GetAnswers"
    images="C:\Program Files\GetAnswers\templates"
    name="GetAnswers"
    objects="C:\Program Files\GetAnswers\objects"
    online="yes"
    query-timeout="30000"
    session-timeout="900000"
    stylesheets="C:\Program Files\GetAnswers\templates"
    templates="C:\Program Files\GetAnswers\templates"
    title="Get.Answers!"
    version="2.0"
    <folder
        id="Getanswers"
        name="Getanswers"
        title="Getanswers"
```

Change the color attribute using the desired color's hexadecimal code.

Change the **title** attribute within a folder tag to the desired text. Don't change the id or name for the folder.

2. Save the changes.

Note: Knowledgebase documents are the only files that display the hit color on key words. Documents such as Word, PowerPoint, and PDF cannot display color on hit key words. For a list of hex color codes, refer to a color chart which you can usually find in any HTML book.

Modifying the Hit List

Using the Answers.xml document in Get.It!, you can modify the following elements of the hit list:

- Hide or show a hit list column
- Include a path name with the title of the document

Hiding a Hit List Column

To hide a hit list column:

1. Open the **Answers.xml** file located in \Program Files\getit\apps\answers\weblication.
2. Locate the column type you want to remove as shown in the example below. You also need to modify the field attributes so that the values are sequential (i.e. h1, h2, h3, h4) for the remaining columns, as shown below:

```
<module name="answers">
  <title> Get.Answers! </title>
  <description image="images/approve.gif" short="Answers"
    long="Answers"/>
  <activities>
    <activity name="search">
      <description short="Search" long="Search"/>
      <forms>
        <!-- Show search results -->
        <form name="search" >
          <title> Get.Answers! </title>
          <instructions> Please enter your question. </instructions>
          <fields>
            <input label="Question" type="text" field="q" size="60"
              required="true" />
            <input type="hidden" field="f" value="hitlist"/>
            <input type="hidden" field="t" value="main-hit-h.htm"/>
            <input type="hidden" field="tf" value="doc"/>
            <input type="hidden" field="tt" value="document-frame.htm"/>
            <input type="hidden" field="x" value="Simple"/>
            <input type="hidden" field="d" value=""/>
            <input type="hidden" field="c" value="redirect"/>
            <input type="hidden" field="s" value="Relevance-Weight"/>
            <input type="hidden" field="ht" value="hitlist.htm"/>
            <!-- These determine the hit list arrangement: -->
            <input type="hidden" field="a" value="Title"/>
            <input type="hidden" field="h1" value="Hit[Hit,3]/>
```

Delete the line shown in bold from the xml file.

Make sure you modify the remaining field values to reflect the column you removed.

```
<input type="hidden" field="h1" value="Relevance-
  Weight[Rank,7]" />
<input type="hidden" field="h2" value="Title[Document,90]" />
</fields>
<actions target-url="http://localhost/lpBin20/lpext.dll">
  <submit> Search </submit>
  <home> Home </home>
</actions>
</form>
</forms>
</activity>
```

3. Save the changes and rebuild the weblocation using wbuild.

Including the Path Name with the Title

To include the path name with the title:

1. Open the **Answers.xml** file located in \Program Files\getit\apps\answers\weblocation.
2. Locate the Title type and replace it with “**Title-Path[Document,90]**”. This will list the titles of a hit document’s parent nodes and the document’s title. Node titles are separated by the forward slash (/). If you change Title to Title-Path, you also need to change the value attribute for the “a” input. The “a” input defines which column will have the link anchors (where a user clicks to open the document). Look at the example below to do this.

```
<module name="answers">
  <title> Get.Answers! </title>
  <description image="images/approve.gif" short="Answers" long="Answers" />
  <activities>
    <activity name="search">
      <description short="Search" long="Search" />
      <forms>
        <!-- Show search results -->
        <form name="search" >
          <title> Get.Answers! </title>
          <instructions> Please enter your question. </instructions>
          <fields>
            <input label="Question" type="text" field="q" size="60"
              required="true" />
              <input type="hidden" field="f" value="hitlist" />
              <input type="hidden" field="t" value="main-hit-h.htm" />
              <input type="hidden" field="tf" value="doc" />
              <input type="hidden" field="tt" value="document-frame.htm" />
```

```

        <input type="hidden" field="x" value="Simple"/>
        <input type="hidden" field="d" value="" />
        <input type="hidden" field="c" value="redirect"/>
        <input type="hidden" field="s" value="Relevance-Weight"/>
        <input type="hidden" field="ht" value="hitlist.htm"/>
        <!-- These determine the hit list arrangement: -->
        <input type="hidden" field="a" value="Title-Path"/>
        <input type="hidden" field="h1" value="Hit[Hit,3]"/>
        <input type="hidden" field="h2" value="Relevance-
            Weight[Rank,7]"/>
        <input type="hidden" field="h3" value="Title-Path[Document,90]"/>
    </fields>
    <actions target-url="http://localhost/lpBin20/lpext.dll">
        <submit> Search </submit>
        <home> Home </home>
    </actions>
</form>
</forms>
</activity>

```

This line defines which column has the link anchors (where a user clicks to open the document).

This line determines what the title looks like in the hit list.

3. Save the changes and rebuild the webapplication using wbuild.



Chapter 6

Adapters

Get.It! ships with several adapters you can use to connect to one or more databases. In addition to the ServiceCenter and AssetCenter adapters that are set up during the Get.It! installation, the following adapters are included with Get.It!:

- JDBC adapter—allows you to establish a full database connection to a database other than the default ServiceCenter and AssetCenter databases.
- LDAP adapter—provides you with a centralized source for information about the people within an organization, eliminating the need to maintain user data in more than one location.
- E-mail adapter—allows you to connect to an external mail server.

This chapter includes detailed information about each adapter. There is also a section at the end of the chapter that contains information about the log files produced by JRun, ServiceCenter, AssetCenter, and the archway connector. These log files can be helpful in troubleshooting connectivity problems.

ServiceCenter Adapter

This section includes information about the `archway.ini` parameters that are specific to ServiceCenter, ServiceCenter event handling, and tips for troubleshooting the Get.It! connection to ServiceCenter.

Archway.ini Parameters

The `archway.ini` file contains parameters that are specific to ServiceCenter. The following table lists the parameters and gives a description of each.

SC Parameter	Description
scadmin	An administrator login must be defined in order to connect to ServiceCenter. The default is <i>falcon</i> , the sample administrator login supplied with ServiceCenter.
scadminpassword	By default, no admin password is required.
scanonymous	With this login, requests sent to archway are processed without going through the Get.It! user interface. Used by the scriptpoller function. The default is <i>falcon</i> .
scanonymouspassword	By default, no anonymous password is required.
schostr	The host name for your ServiceCenter installation. The default is <i>localhost</i> .
scport	The port number for your ServiceCenter installation. The default port number for a full client is <i>12670</i> .
sclog	The location of the <code>sclog</code> file. The default is <code>C:\Program Files\getit\sc.log</code> .

ServiceCenter Event Handling

The ServiceCenter adapter has been enhanced to provide a detailed `eventin` record that will give status when an event fails to execute or fails to create an eventout response.

The following is an example of an `eventin` record. The record was generated because an illegal approver tried to send an approval event:

```

<eventin>
  <evtype>approval</evtype>
  <evtime></evtime>
  <evsysseq>3991blff0036f001</evsysseq>
  <evusrseq>3991blff0036f001</evusrseq>
  <evsysopt></evsysopt>
  <evuser>Hartke</evuser>
  <evpswd></evpswd>
  <evsepchar>^</evsepchar>
  <evfields>^ocmq^Q1102^approve^^Hartke^^^</evfields>
  <evexpire>0</evexpire>
  <evstatus>error</evstatus>
  <evnumber></evnumber>
  <evlist></evlist>
  <evtimestamp>2000-08-09T19:40:05+00:00</evtimestamp>
  <evcount></evcount>
  <evnetnm></evnetnm>
  <evcode></evcode>
  <evmsg>
    <entry>Cannot execute application: es.approval</entry>
    <entry>Unrecoverable error in application: es.approval on
    panel decide.exit</entry>
    <entry>You are not authorized to access Request Mgmt Quotes.
    </entry>
  </evmsg>
  <evid></evid>
  <sysmodcount>1</sysmodcount>
  <sysmoduser>N/A</sysmoduser>
  <sysmodtime>2000-08-09T19:40:05+00:00</sysmodtime>
</eventin>

```

The Archway script can use this information as necessary. For instance, it may display the `evmsg` error messages returned by the RAD application, or messages indicating that an event has failed because of customized Format Control rules.

Using the `_event` parameter

The “`_event`” parameter can be used in a script to specify the name of the event to be used in an `SCDocManager` operation.

If an `_event` parameter is not defined, the standard insert or update attributes of the document are used. Refer to “ServiceCenter-Specific Attributes” in Appendix B for information on the insert and update attributes.

The following is an example of the use of the `_event` parameter:

```

var msgTicket = new Message( "Problem" );

...
msgTicket.set( "_event", "epmc" );
// Tell SCDocManager to use an epmc event for this update

```

```
archway.sendDocUpdate( "sc", msgTicket );
```

These lines will override the default insert and update attributes and instruct the SCDoc-Manager to use the attributes defined in the script instead.

Troubleshooting the ServiceCenter Database Connection

If you are having trouble making a connection between Get.It! and the ServiceCenter database, verify the following:

1. Check the Control Panel page in the Admin module to confirm the database connectivity status. See “Verifying Adapter Connections” on page 5-24.
2. If “sc” is disconnected, verify that the ServiceCenter service is running (the ServiceCenter console has been started).
3. If the ServiceCenter service was not running and you have started it, also restart JRun after the connection is established.
4. Verify that you have ServiceCenter *full* client connectivity by starting a client that is pointed to the port listed in the `archway.ini` file.

AssetCenter Adapter

This section includes information about the `archway.ini` parameters that are specific to AssetCenter, tips for troubleshooting the AssetCenter connection, and tells how to set AssetCenter feature links.

INI Parameters

The following table lists the `archway.ini` AssetCenter parameters and gives a description of each.

AC Parameter	Description
<code>acadmin</code>	An administrator login must be defined in order to connect to AssetCenter. The default is <i>Admin</i> , the sample administrator login supplied with AssetCenter.
<code>acadminpassword</code>	By default, no admin password is required.
<code>acanonymous</code>	With this login, requests sent to archway are processed without going through the Get.It! user interface. Used by the scriptpoller function. The default is <i>Admin</i> .
<code>acanonymouspassword</code>	By default, no anonymous password is required.
<code>acdatabase</code>	The name of the AssetCenter database to which Get.It! will connect. The default is <code>ACDemo350ENG</code> .
<code>acdateformat</code>	Establishes the date format for AssetCenter. The default is <code>yyyy-MM-dd</code> .

AC Parameter	Description
acapidll	<p>Used with AssetCenter version 3.5 and above. This parameter is not mandatory, since Get.It! is able to detect which DLL to load and where to find it, based on the connection name defined by the acdatabase parameter.</p> <p>When this parameter is not specified, Get.It! uses the information provided by AssetCenter in the connection description (in amdb.ini), to determine the DLL name. Use this parameter only if the default name determination is not successful.</p>
acapidllpath	<p>Gives the path to the folder in which the AssetCenter API DLL is located. If this parameter is left blank, Get.It! will search the folders in which AssetCenter is installed for a DLL with the expected name. If the requested DLL is not found, Get.It! will load it from its bin folder.</p> <p>This parameter should be used only if the default path determination does not work as expected. Note that in the case where the locale parameter in archway.ini contains “ja” (meaning that you intend to use Japanese), Get.It! will only load the API DLL from an AssetCenter Japanese folder, and if it does not find it there, it will load it from bin/ja.</p>
acdefaultloginclass	<p>Establishes the type of default user login. This parameter can have four different values:</p> <ul style="list-style-type: none"> • If it is empty or not specified in archway.ini, the self-registered users are added to AssetCenter with the default login class defined in AssetCenter. Out of the box, the default for a registered user in AssetCenter is named user. • If the value is <i>casual</i>, (acdefaultloginclass=casual) the self-registered user will be added as a casual user. • If the value is <i>floating</i>, (acdefaultloginclass=floating) the self-registered user will be added as a floating user. • If the value is <i>named</i>, (acdefaultloginclass=named) the self-registered user will be added as a named user. <p>Casual user, floating user, and named user are described in the AssetCenter documentation.</p>

Troubleshooting the AssetCenter Database Connection

If you are having trouble making a connection between Get.It! and the AssetCenter database, verify the following:

1. Check the Control Panel page in the Get.It! Admin module to confirm the database connectivity status.
2. If “ac” is disconnected, verify that the “acdatabase” parameter in the `archway.ini` file is the same as the database name displayed when you log into AssetCenter. For example, in the system as shipped, the acdatabase parameter is set to ACDemo300ENG. Refer to your AssetCenter documentation for assistance with login procedures.
3. Verify that all AssetCenter settings match Get.It! settings. Log into the AssetCenter database. Make sure that the login account referenced in the Get.It! settings matches the login for AssetCenter. Also verify that Get.It! is using the correct user name and password for the connection. You can do this by selecting File | Manage Connections in AssetCenter.
4. Check the Get.It! ODBC connections to AssetCenter. Depending on the way you run JRun, it will look for either an ODBC User DSN or an ODBC System DSN.
 - If you start JRun as a service (the usual method), it references the System DSN for the ODBC connection to the AssetCenter database.
 - If you start JRun as an application, it references the User DSN to determine the ODBC connection.

AssetCenter Feature Links

The following is a JScript function to set feature links, a solution for an AssetCenter limitation.

```
//general case assetcenter feature link population.  
//to be used when a feature link is defined to a  
//table in assetcenter. you must link with multiple  
//tables to link with the required record.  
  
//expects many parameters: amFeature table, SQLName,  
//amFeature, TargetTableName, field name of ID field  
//in target table.  
  
function featureLink(  
  
    //amFeature SQLName  
    sFeatureSQLName,  
  
    //table that Feature points to
```

```

        sTableName,

//ID field in sTableName
        sld,

//string value of destination record
        sDestinationValue,

//ID of destination record of link
        IDestinationId
    )
}
    var msgFeature;
    var strFeatId;
    var msgFeatureRelative;

//locate the amFeature record
    msgFeature = archway.sendQuery("ac",
        "SELECT IFeatId FROM amFeature WHERE SQLName = "`
        + sFeatureSQLName + "`",
        0,1);

//convenient storage for the Id
    strFeatId = msgFeature.get("IFeatId");

//locate the record in sTableName identified via the
//amFeature record
    msgFeatureRelative = archway.sendQuery("ac",
        "SELECT IFeatValld FROM" + sTableName +
        "WHERE" + sld + "=" + msgDestinationRecord.get("Id")+
        "AND IFeatId = " + strFeatId,
        0,1);

//modify the record in sTableName with data from
//msgDestinationRecord
    msgFeatureRelative.set("fval", IDestinationId);
    msgFeatureRelative.set("ValString", sDestinationValue);
    archway.sendUpdate("ac", msgFeatureRelative);

    return;
}

```


You can call this to set feature link values. The following example adds a “Manager” feature link to the amemplDept table to link an employee to a manager.

```
featureLink(  
  
    //SQLName of the amFeature entry  
    "Manager",  
  
    //feature table associated with feature added to "amEmplDept"  
    "amFvEmplDept",  
  
    //name of ID field in amEmplDept  
    "IEmplDeptId",  
  
    //string value of the manager record (e.g., Name, FirstName)  
    sManagerName,  
  
    //ID of destination record of link (the "manager" we're linking with)  
    IManagerId);
```

JDBC Adapter

The Java Database Connectivity (JDBC) adapter allows you to create a connection between Get.It! and a third-party database.

You can set up as many JDBC adapters as you need. The only limitation is that each adapter must point to a different database target string.

Adding and configuring a JDBC adapter requires several steps:

- Creating the connection between Get.It! and the database.
- Validating the connection.
- Adding the database settings to the Settings form.
- Creating a module in the portal so that information from the database will be displayed in the desired format.

The following procedures outline these steps, using the example of establishing a connection to a database that stores data about employees.

Adding a JDBC Adapter

There are two steps to adding a JDBC Adapter to Get.It!:

- Verify the Data Source Name (DSN) for the database to which you want to connect.
- Update the `archway.ini` file with information about the database.

Verifying the System DSN

The first step in adding the JDBC adapter is to verify the DSN for your database. You will need to add this information to the `archway.ini` file.

If you have not yet created a system DSN for your database, use the following procedure to do so.

1. Open the Control Panel (Start->Settings->Control Panel) and double-click the ODBC Data Sources icon.
2. Select the System DSN tab, and then click **Add**.

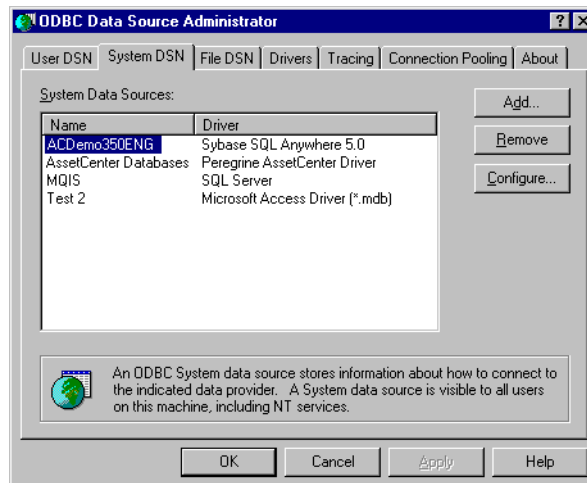


Fig. 6.1 ODBC Data Source Administrator

3. Select the driver for which you want to set up a data source. For this example, we are using the Microsoft Access Driver, because the example database was created using Microsoft Access.

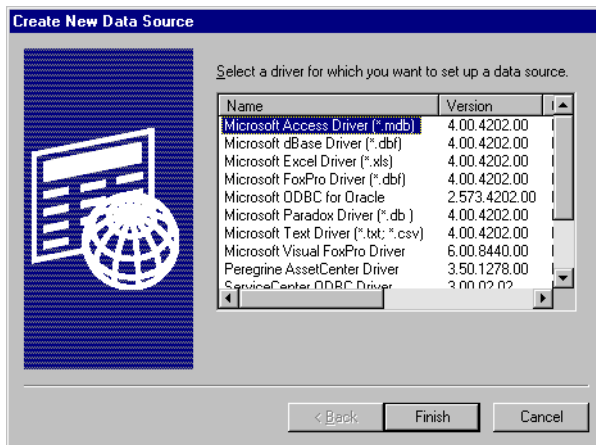


Fig. 6.2 List of available drivers

4. Click **Finish**.
5. In the dialog box displayed, give the data source a name. You can name it anything you want. For this example, we have used “Access_JDBC_test”.

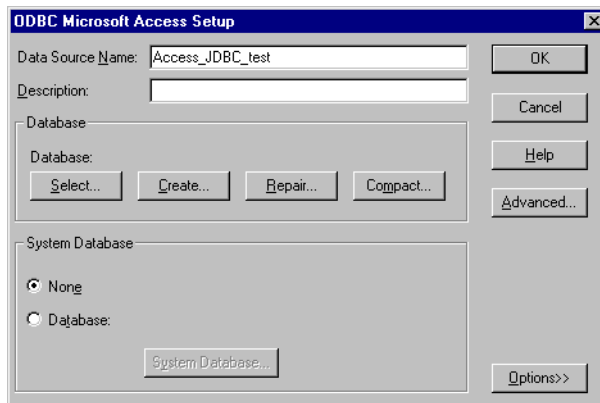


Fig. 6.3 ODBC setup—data source name

6. Click **OK**. The new DSN is added to the list of data source names.

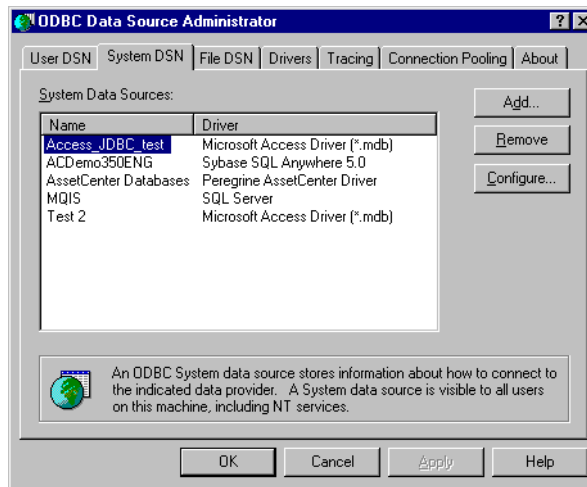


Fig. 6.4 New system DSN added

7. Click **OK**.

Updating the Archway.ini File

1. Open the `archway.ini` file in the `C:\Program Files\getit\bin\` directory.
2. Update the adapters line with the JDBC Adapter, so the line would look something like:

```
adapters=sc=SCAdapter;portalDB=SCAdapter;xx=JDBCAdapter
```

where xx is a two character designation of the database to which you are linking.

3. Add the following lines to the `archway.ini` file. Replace xx with the two character designation of the database to which you are linking. Replace `<odbc name>` with the system DSN for your database.

```
xxdatabase=<odbc name>
xxdatabaseurl=jdbc:odbc:
xxdatabase driver=sun.jdbc.odbc.JdbcOdbcDriver
xxcasesensitive=false
xxanonymous=
xxanonymouspassword=
xxadmin=
xxadminpassword=
```

For this example, the modified lines would look like this:

```
madatabase=Access_JDBC_test
madatabaseurl=jdbc:odbc:
madatabasedriver=sun.jdbc.odbc.JdbcOdbcDriver
macasesensitive=false
maanonymous=
maanonymousspassword=
maadmin=jsmith
maadminpassword=test
```

The fields can be updated after you have changed the Administration module weblication to display them. The “xxadmin” and “xxadminpassword” are used to actually connect to the database and, therefore, must be a valid database user.

The “xxanonymous” and “xxanonymouspassword” as well as names used in the login screen are used for Get.It! user sessions.

The “xxdatabaseurl,” “xxdatabasedriver,” and “xxcasesensitive,” and anonymous login fields are optional. All others are required.

4. Save the file.

Validating the JDBC Adapter Connection

Once you have completed the steps for adding an adapter, you can verify that the connection has been made.

1. Go to the database you want to query and find a table against which you can run a query. Make note of the table name.
2. Type the following query in the address field of your Web browser, and then press Enter:

```
<hostname>\servlet\archway?<adapter name>.query&_table=<table name>
```

where <name of adapter> is the two-letter designation you gave your database in the `archway.ini` file.

For our example, we used a table called “Employees” in the sample database, so our query would look like this:

```
localhost\servlet\archway?ma.query&_table=Employees
```

3. In the dialog box displayed, select to open the file from its current location, and then click **OK**.
4. Select a text editor to display the data, and then click **OK**.

Figure 1.5 shows an example of data displayed using Notepad.

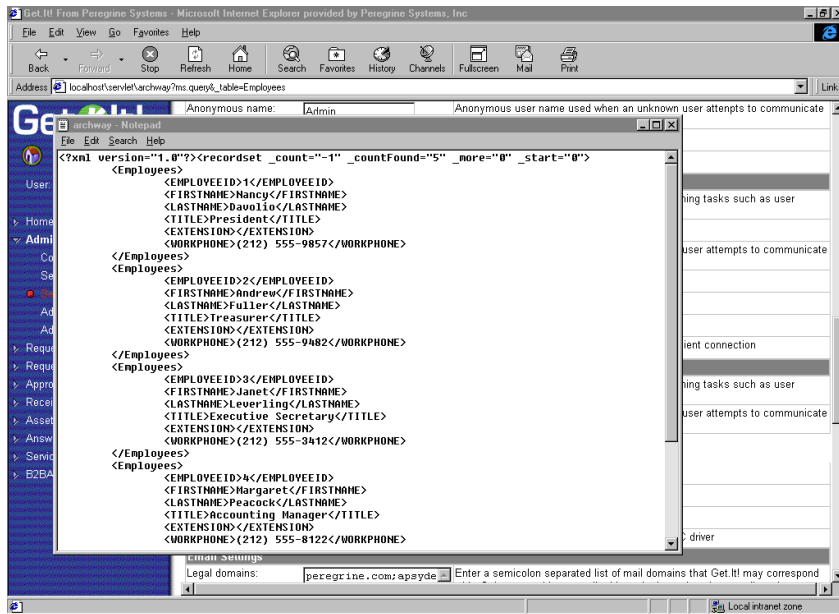


Fig. 6.5 Verification that the JDBC adapter is processing data

Adding the New Database Settings

The Settings form in the Get.It! Admin module contains the settings for the ServiceCenter and AssetCenter database connections.

You will need to update the `admin.xml` file to add the JDBC adapter settings to this form.

Updating the Admin.xml File

1. Open the `admin.xml` file from the `C:\Program Files\getit\apps\common\weblication\` directory.
2. Find the activity called “Settings.”
3. Scroll down until you see where the AssetCenter and ServiceCenter adapters are set up.

Enter the following lines after the AssetCenter and ServiceCenter adapter information. Be sure to update the “JDBC” and “xx” with the information for the system to which you are linking. Where “xx” is used, replace uppercase “XX” with uppercase letters and lowercase “xx” with lowercase letters:

```
<section label="JDBC (xx) Adapter Settings">
  <entry>
```

```

<input field="XXAdmin" type="text" label="Administrator name"
  size="20" />
<instructions>Administration user used by Get.It! when performing
tasks such as user authentication and registration</instructions>
</entry>
<entry>
  <input field="XXAdminpassword" type="password"
    label="Administrator password" size="20" blank="_blank" />
  <instructions>Administration password</instructions>
</entry>
<entry>
  <input field="XXAnonymous" type="text" label="Anonymous name"
    size="20" />
  <instructions>Anonymous user name used when an unknown user
  attempts to communicate through archway</instructions>
</entry>
<entry>
  <input field="XXAnonymouspassword" type="password"
    label="Anonymous password" size="20" blank="_blank" />
  <instructions>Anonymous user password</instructions>
</entry>
<entry>
  <input field="XXDatabase" type="text" label="DataSource"
size="20" />
  <instructions>Data source name for ODBC driver</instructions>
</entry>
<entry>
  <input field="XXDatabaseUrl" type="text" label="JDBC Url"
size="20" />
  <instructions>Url for JDBC driver (optional)</instructions>
</entry>
<entry>
  <input field="XXDatabaseDriver" type="text" label="JDBC Driver"
    size="20" />
  <instructions>Alternate JDBC driver (optional)</instructions>
</entry>
<entry>
  <input field="XXCaseSensitive" type="checkbox" label="Case
  Sensitivity" value="true" valueoff="false" />
  <instructions>Select this to toggle case sensitivity in the JDBC
  driver</instructions>
</entry>
</section>

```

4. Save the file in your `... \user` directory. If you save it in the `common` `Get.It!` directory it will be overwritten the next time you load a new version of `Get.It!`
5. Run `wbuild getit`. See Chapter 4 for detailed instructions on running the `wbuild` command.
6. Verify that the fields have been added to the Settings page.

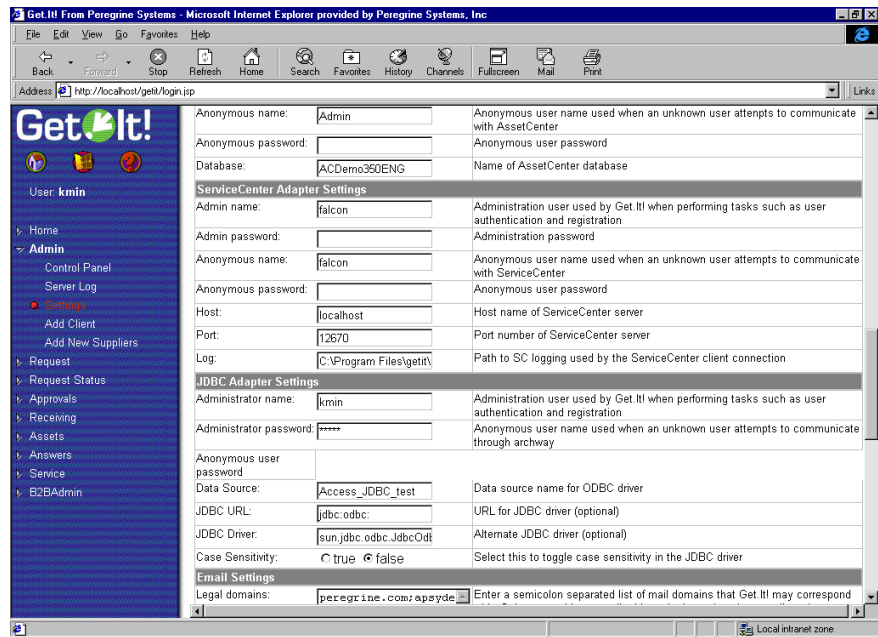


Fig. 6.6 JDBC adapter settings added to the settings page

Creating an Interface to the Database

In order to display formatted data from your database or to add new information from Get.It! to the database, you will need to add a new module to the Get.It! portal. This process includes the following three steps:

- Modify an XML file to establish how you would like the data to be formatted.
- Modify schemas to map to the fields in your database tables.
- Modify the JavaScript to point to your database.

All of these procedures can be done by copying existing files and changing them to pull data from your database and format it. The files you will need are located in the `...getit\apps\common` directory. Copy the files you want to use into the equivalent folders in the `...getit\apps\user` directory so that they will not be lost when you upgrade to a later version of Get.It!. See “Adding a New Module” on page 4-25 for instructions for adding a module to the portal.

Preventing Lost Connections

If you experience lost connections while using the JDBC adapter, you can use a polling script which will periodically check for database connectivity and reconnect when a connection is lost. The script is then run through `scriptpoller.ini` on a periodic basis.

The following script (`pingDB.js`) is an example. You will need to customize this script for your own database name and queries as noted.

```
//-----  
// PingDB - attempts a well known query periodically, on error  
// reconnects the JDBCAdapter to the database using the "_connect"  
// event which will attempt a disconnect and then reconnect.  
//  
// Two functions are defined:  
// start() - executes exactly once  
// run()   - executes on the polling interval  
//  
//-----  
Archway = Packages.com.peregrine.archway;  
Message = Archway.Message;  
  
//-----  
// Start function .. can build parameters for run method  
//-----  
function start( msg )  
{  
    var msgRet = new Message();  
    msgRet.set("message", "ok");  
    return msgRet;  
}  
  
//-----  
// run function .. subsequent invocations  
//-----  
function run( msg )  
{  
    var msgDB = new Message("test_query");  
    var msgRes = null;  
    var msgRet = new Message();  
  
    msgDB.set("query", "select * from tab");  
    msgDB.set("_count", "1");  
  
    msgRes = archway.send( "xx", "query", msgDB );  
    if (msgRes.get("message"))  
    {  
        msgDB = new Message("_connect");  
        msgRes = archway.sendEvent( "xx", msgDB );  
    }  
}
```

Replace this query with one that you have used with your database and you know will succeed every time.

Replace "xx" with the two-letter designation of the target database identified in the `archway.ini` file.

```

    msgRet.set("message", "ok");

    return msgRet;
}

```

Calling a Stored Procedure

Stored procedures can be called with input and output parameters. When the procedure is executed, the adapter first attempts to get any result sets that may have been returned, then attempts to retrieve the set of parameters with which the function was called. The resulting parameter list includes both input and output parameters and their values.

All parameters are treated as (and assumed to be declared as) Varchar or String type, with other data types to be supported in a future release.

Input and output parameters are specified with the "type" attribute; valid settings are "IN/OUT/INOUT".

There are three ways to call a stored procedure:

- From the Web browser through CGI syntax in a URL.
- From a JScript file or post stream with an XML document containing `_call` and `_sql` tags.
- Using the Get.It! Document Manager.

The following is an example using a stored procedure called "insertEvent()" in an MSSQL7.0 database with two parameters; the first is of type "input" and the second, "output". When called, the procedure verifies the input data and inserts a row returning the new row ID number.

Using a URL

```

http://localhost:8080/prgn/servlet/archway?ja.event
&_sql={call insertEvent('User1',?)}&name=User&value=User1&type=IN
&name=Id&type=OUT

```

Using a JScript

```

function processInsert( msg )
{
    var msgEvent = new Message("event");
    var strCall = "{CALL insertEvent('User1',?)}";

    msgEvent.set("_sql", strCall);
    msgArg = new Message("_arguments");

    elTag = msgArg.add("name", "UserName");
    elTag.setAttribute("type", "IN");
    elTag.setAttribute("value", "User1");
}

```

```

    elTag = msgArg.add("name", "Id");
    elTag.setAttribute("type", "OUT");

    msgEvent.add(msgArg);

    msg = archway.sendEvent("ja", msgEvent);

    return msg;
}

```

Using a Document Insert Or Update

This example uses the following schema definitions:

```

<?xml version="1.0"?>

<schema>

<!--=====
Generic Schema Definitions
=====-->

<documents name="base">
  <document name="ProcEvent">
    <attribute name="UserName" type="string"/>
    <attribute name="Id" type="id"/>
  </document>
</documents>

<!--=====
JDBCAdapter Schema Definition
=====-->

<documents name="ja">
  <document name="ProcEvent" table="None" insert="insertEvent"
update="updateEvent">
    <attribute name="UserName" field="NAME" insertEvent="IN"
updateEvent="IN"/>
    <attribute name="Id" field="ID" insertEvent="OUT"
updateEvent="IN"/>
  </document>
</documents>

</schema>

```

using the following JScript function:

```

function processInsert( msg )
{
    var msgRequest = new Message( "ProcEvent" );

```

```
msgRequest.add( "UserName", "User1" );
msgRequest = archway.sendDocInsert( "ja", msgRequest );

return(msgRequest);
}
```

All methods above will produce the following output:

```
<?xml version="1.0" ?>
<_doc>
  <_arguments>
    <Id>915326</Id>
    <UserName>User1</UserName>
  </_arguments>
</_doc>
```

LDAP Adapter

Note: LDAP functionality is not available on MVS systems.

Lightweight Directory Access Protocol (LDAP) directories provide a centralized source for information about the people within an organization. E-mail addresses, telephone numbers, fax numbers, user IDs, and passwords can be defined in the LDAP directory and referenced by various applications, eliminating the need to maintain user data in more than one location.

The use of an LDAP directory allows users without ServiceCenter operator records to log on to Get.It! by providing minimal login information, such as login ID and password.

The information in this section assumes prior knowledge of the general setup and configuration of LDAP, and ability to write the associated JScript.

Connecting LDAP to Get.It!

There are three procedures that need to be completed in order to connect to Get.It! using LDAP:

- Update the `archway.ini` file to include the LDAP parameter.
- Create a JavaScript to query Get.It!.
- Reset the server.

This section provides information on updating the `archway.ini` file. Refer to your LDAP documentation for instructions on setting up an LDAP adapter. Sample scripts are not provided with Get.It! because of the variety of possible configurations. However, Chapter 4 of this guide includes instructions for modifying JScripts.

Updating the archway.ini file

To configure the LDAP adapter, update the `archway.ini` file with the following information:

1. Add the adapter name to the adapters line:

```
adapters=sc=SCAdapter;ac=ACAdapter;ldap=LDAPAdapter
```

2. Define the LDAP connection parameters:

```
ldapurl=xxx  
ldaplogin=xxx  
ldappassword=xxx
```

Replace the “xxx” with your system and user-specific information.

E-mail Adapter

The E-mail adapter can be set up using Java or from an Archway ECMA script. The various calls that are available are the same either way. The E-mail adapter is based on the Sun `Javax.mail` classes for mail, especially:

- `javax.mail.Session`
- `javax.mail.Message`
- `javax.mail.Transport`
- `javax.mail.search`

and several other related classes. Detailed documentation on these is available from Sun's `java.sun.com` site.

The following script illustrates how the E-mail adapter can be used to connect to a mail server and how a given mailbox at that server can be processed:

```
// FESI definitions to point to the Archway mail adapter Java classes  
Archway = Packages.com.peregrine.archway;  
MailAdapter = Archway.adapters.MailAdapter;  
  
// Allocate a new MailAdapter object  
var ma0 = new MailAdapter();  
  
// Optional: Turn on Javax.mail debug mode by calling  
// "setDebug( true )"  
// If this is done, copious javax.mail debug messages will be written  
// to JRUN standard output log  
ma0.setDebug( true );  
  
// Connect to a specified mailbox at a specified IMAP4-capable
```

```

// mail server, passing the hostname of the server, a userid,
// and a password
ma0.setConnection( "exchange.mycorp.com", "joesmith",
"opensesame" );

// If we could not connect for some reason, print MailAdapter last
// error in the log and return false
if ( ma0.connect() != 0 )
{
    var err = ma0.getLastErrorMessage();

// This returns an Archway error message containing last javax.mail
// error info
    env.error( "Got error: " + err.getMessage() +
"trying to connect to mailbox" );
    return false;
}

// Get a logical Archway lock on the mailbox we are going to process
var strMailLock = "MailAdapter:" + "joesmith"
env.getLock( strMailLock );

// Build the SQL query to query the mailbox
var mailQuery = "SELECT * FROM INBOX ";
if ( docType != null )
{
    mailQuery += " WHERE subject = '" + docType + "' ";
}
if ( lastRecvDate != null && lastRecvDate != "" )
{
    mailQuery += " AND receivedDate > " + lastRecvDate ;
}

// Issue the query. Ask for 4 messages starting with zeroeth message.
// Specifying -1 for the count retrieves all messages
var m = ma0.doQuery( mailQuery, 0, 4 );

// Make sure it worked
if ( m == null )
{
    env.error( ma0.getLastErrorMessage() );
    env.releaseLock( strMailLock );
    return false;
}

env.debuglog( "Response from mailadapter was: " + m.getContent() );
var i = 0;
var docsProcessed = 0;
var list = m.getList("message");
for ( i=0; i < list.getLength(); i++ )

```

```

    {
// Get the mail message represented by this list entry
    var mm = ma0.getMailMessageByNumber( list.get( i,
"messageNumber" ) );
    if ( mm == null )
    {
        env.log( "Failed to retrieve mail message by number - " +
ma0.getLastErrorMessage() );
        env.releaseLock( strMailLock );
        return false;
    }
    env.debuglog( "Mail message " + i + " is: " + mm.getContent
( false ) );

// Get a few elements out of the Mail message
    var msgID    = mm.get( "messageID" );
    var msgNo    = mm.get( "messageNumber" );
    var text     = mm.get( "text" );
    var subject  = mm.get( "subject" );
    var dtReceived = new Date( mm.get( "receivedDate" ) );

// Delete the Mail message
    ma0.deleteMailMessageByNumber( msgNo );
    }

// Commit inbox folder changes. This does a Java mail expunge
// operation
    ma0.commitInboxFolderChanges();

// Release lock on mailbox
    env.releaseLock( strMailLock );

// Close the mail box
    ma0.disconnect();

```

Verifying Adapter Connections

The status of a connection from Get.It! to a database can be verified from the Control Panel in the Admin module. In the example in figure 6.7, “sc” and “getanswers” are shown as disconnected because the ServiceCenter console was not started before logging into Get.It!.

Control Panel

ERROR: Unable to reconnect to SC. Please verify that the SC server is up and restart Archway.

Here is a list of the adapters currently registered in this server. You may click on any adapter to find out further details about the connections. If necessary, you may also reset the server and all its connections.

Archway version timestamp: Get.It! MAIN DEV BRANCH, Build 200009191221

Connection Status:

Target	Adapter	Status
portalDB	com.peregrine.archway.adapters.ACAdapter	connected
getanswers	com.peregrine.archway.adapters.SCAdapter	disconnected
ac	com.peregrine.archway.adapters.ACAdapter	connected
ma	com.peregrine.archway.adapters.JDBCAdapter	connected
sc	com.peregrine.archway.adapters.SCAdapter	disconnected

[Reset Server](#)

Fig. 6.7 Verifying adapter connections.

For more information on connectivity troubleshooting with each adapter, refer to the appropriate section on each Get.It! adapter provided in this chapter.

Log Files

There are a number of log files produced by JRun, ServiceCenter, AssetCenter, and the archway connector that provide information that can help you troubleshoot your Get.It! installation, including connectivity problems.

The following table lists the paths to the log files and gives a description of the types of messages written to each file.

Path to Log File	Description
\\JRun\\jvm-default\\logs\\stdout.log	Standard output from JRun, including “writeIn” and Log.log statements from Jscript.
\\JRun\\jvm-default\\logs\\event.log	A log of events: initialization of objects, need to re-initialize objects after modifications, and so forth.
\\JRun\\jvm-default\\logs\\stderr.log	Standard Error output from JRun. Includes Java exceptions and similar errors.
\\JRun\\jvm-default\\services\\jse\\logs\\error.log	Logs information similar to stderr.log above.
\\JRun\\jvm-default\\services\\jse\\logs\\event.log	Logs information similar to event.log above.
archway.log	The location of this log file is specified in the archway.ini file. The information included in the log file is also determined by the INI file, but includes things like env.log , env.debuglog , and query information.
\\Program Files\\ServiceCenter\\sc.log	Information regarding the ServiceCenter connection.
\\Program Files\\AssetCenter\\ac.log	Information regarding the AssetCenter connection.



Chapter 7

Localization Support

Make sure the Locales and Currencies fields in the Administration Module Settings are set correctly before you begin localizing files.

Get.It! allows each user to dynamically select the language in which they would like the Get.It! windows displayed. This is supported through language-specific string files which you can update if you are adding your own strings.

General Localization Steps

Before you can localize the strings you create, complete the following two steps:

1. Use the Administration Module to update the Settings so the “Locale” field includes the languages you want to have available. Use the two-character language code defined in the ISO standard. You can find a chart of the ISO codes in Chapter 2, “Get.It! Administration Module,” of the *Get.It! Administration Guide*.
2. Update or create your modifications to Get.It! just as you always would. If needed, refer to Chapter 4, “Tailoring Get.It!,” for the steps to complete.

The following is a general overview of the steps you will do to translate your modifications.

3. Run `wbuild getit`.
4. Run `strbuild getit`. This will update your weblication string files.
5. After you run `strbuild`, look in these files and update all empty quotes with the correctly translated strings. Search for quote quote (“ ”) to find all the empty strings.
6. Log into Get.It! When Get.It! attempts to write a screen, it first looks for a string in a country-specific file. If a string is not found in a country-specific file, Get.It! then looks in the language-specific file.

See “</form>” on page 4-15 for an step-by-step example of this process.

You can further define language within regions by creating STR files defining only those strings which are different and adding the country name the file names. For example, the English extract file for Great Britain could be `getit_en_GB.str`.

Archives

A copy of all STR files is stored in an archive file. This archive file is called `...getit\apps\getit.zip`. You can transmit this file to a third party translation service if you do not want to update the STR files yourself.

Externalizing ECMAScript Messages

The ECMA scripts defined in each Get.It! package also contain some messages. For example, the following is a script that sends a message to the user:

```
User.addMessage( "You have no open problem tickets" );
```

Messages like these are externalized by storing them in a file named `<package>.en.str` found in each package folder. For example, messages belonging to the Get.Service! package are stored in `...getit/apps/service/service_en.str`.

The externalization process is as follows:

1. During development, all messages should be placed in a STR file. For example:

```
statusNoTickets, "You have no open problem tickets"
```

2. Instead of including the message in the ECMA script, use the API interface shown below:

```
User.addMessage( IDS.get( "service", "statusNoTickets" ) );
```

The `IDS.get()` API requires three parameters. The first is package name where a message is defined. The second is the string ID defined for the entry. The user's preferred language is determined by the ScriptRunner.

ID names must be unique within a package. In addition, each ID should be prefixed by the name of the script file where it is associated.

3. When ready to localize, open the appropriate string file. Save the file in the same user directory where you saved the script file you updated as `user_languagecode.str` where `languagecode` is the two character code for the language (`user_fr.str` for French).

At runtime, Archway picks up messages from the appropriate STR file as determined by the user session language preferences. Thus, one user could view English messages while other views them in French at the same time.

Externalizing ECMAScript Messages with Variables

The IDS interface includes methods for replacing variables into a message. For example, consider the following message

```
"Asset " + strOld + "will be replaced by asset " + strNew
```

Such a message should be externalized as follows:

```
ID, "Asset %1 will be replaced by asset %2"
```

This allows translators to place the %1 and %2 replacement tags anywhere that makes sense according to grammatical rules of the target locale.

The IDS interface supports the %1 tags with calls such as:

```
IDS.get( user, strID, str1, str2, ..., strN );
```

Where str1, str2, ... strN respectively replace %1, %2, ... %N

IDS defines functions for up to three replacement strings. For greater numbers of replacements, use the get() function that takes a string array.

Externalizing Messages in XSL Templates

While it is rare, the Get.It! XSL templates define some messages of their own. For example, the label for a “back” button is hard coded in the XSL templates, providing a uniform label anywhere a <back> weblication element is found.

There are two ways XSL templates externalize strings. The first is by using the \$\$IDS() sequence, as shown in the following example:

```
<xsl:template name="genBackButton">
<input type="button" value="$$IDS(common,xslBack)">
</xsl:template>
```

The XSL above refers to a string defined in the common module’s message STR file:

```
xslBack,          "Go Back"
```

The \$\$IDS() sequence is useful when embedding IDS strings in generated HTML. However, some XSL code may actually generate JSP scriptlet code. In such cases, use the normal IDS APIs already described above for Java code externalization. For example:


```
<jsp:scriptlet>
  if ( user.getName().length() > 0 )
    out.println( IDS.get(user, "portal", "xslWelcome") +
      user.getName() );
</jsp:scriptlet>
```

The code above retrieves anIDS string defined in the portal module.

Japanese Locale

You must run the wbuild command when running the Japanese version or locale of Get.It! To run the Japanese version you must type the correct locale in the Locales field. You can find this field by clicking on the Admin module and then scrolling to the Locales field. In this field type “ja.” Click the Save button, located at the bottom of the screen.

Because Japanese characters are not presently supported under the ISO-8859-1 character set, you must run the wbuild command. All character sets that fall outside of theISO-8859-1 character set require the wbuild command to be executed.



Chapter 8

Troubleshooting

Weblication

Q: How do I include links in a weblication form that can lead to different queries executed and displayed on the resulting form?

A: Let's say that you want two links on a page: One that lists Open tickets and another that lists Closed tickets.

Your form might look like the following:

```
<form name="search">
  <title> Search tickets </title>
  <fields>
<link target-form="list" param="Status=open"> Open Tickets </link>
<link target-form="list" param="Status=closed"> Closed Tickets </
link>
  </fields>
</form>
```

Both of these links will call the "list" form. The "list" will have an `onload` script that is called and receives the value passed in the `param` attribute of the links. When you click on *Open Tickets*, the `Status=open` values are passed on to the script and subsequently used by the script's queries.

Q: Is it safe to use a global script variable in FESI script running within the Archway environment? The global would be private to the thread but wider in scope. Is there any possibility of two threads colliding into the variable?

A: The way to maintain session-specific data is to store it in the `user` object. Each logged in user has their own user object. You can use something similar to the following script:

```
user.set( "name", "value" );
and
value = user.get( "name" );
```

Q: I want to obtain details about the Requester into the PurchaseOrder document. `amPOrder` has a link field to `amRequest`, and `amRequest` has a link field to `amEmpDept`. What is the schema syntax for creating the "double" link so that I can get, for

example, Requester.Email into the PurchaseOrder document?

A: If you want to display the requester's email as a field of a PurchaseOrder then you can try including an attribute such as the following:

```
<attribute name="RequesterEmail"
field="Request.Requester.Email"/>
```

If you want to include the entire Requestor record instead of having EMail as a Purchase-Order field, you can add nested Documents to the PurchaseOrder document:

```
<document name="Request" ...>
<document name="Requester" ...>
<attribute name="EMail" ...>
```

Q: How do you format a non-editable field?

A: Do not assign a `type="text"` tag. For example,

```
<field label="Ticket number" field="Id"/>
<field label="Category" field="Category"/>
```

Q: How do you adjust Listbox size? The rows and cols and size do not work.

A: You cannot control table width. Height depends on the number of elements in the result set. You can set the `rows` attribute but it is maximum value. If the result set has more then you can use *Next* and *Prev* buttons.

Q: Is there a method for determining current user and password?

A: Yes.

```
user.getName() or user.get( "_name" ) and user.get( "_password" );
```

Q: How can I test a script?

A: Assuming, for example, that you have two script functions in a script file named `script.js`:

```
function getTicket( msg )
{
  return archway.sendDocQuery( "sc", "Problem", msg );
}
```



```
function newTicket( msg )
{
  var msgProblem = new Message( "Problem" );
  msgProblem.addChildren( msg );
  return archway.sendDocInsert( "sc", msgProblem );
}
```

You can test these in a browser by hitting Archway using either (`target.command`) or (`object.method`), as in the following.

```
http://webservername/servlet/archway?script.test
```

Q: How do I process input data entered by a user in a form containing a table. For example, the Get.Resources! catalog screen contains a table of products, and each row has a spinner field that allows the user to enter a count value. How are these count values submitted to an archway script, and how should they be processed?

A: Archway defines APIs to, thereby simplifying this process. The first step is to define the table in a weblocation. The following is sample code that defines the product catalog tables:

```
<table record="Product" rows="10">
<link target-form="product" field="Id"/>
<column label="Count" field="nCount" type="spinner" key="Id"
size="3"/>
<column label="Brand" field="Brand"/>
<column label="Model" field="Model"/>
<column label="Price" field="Price"/>
</table>
```

Notice the definition of the "Count" column. It defines a column of spinner fields. This column contains two important attributes:

Attribute	Function
field="nCount"	States that the column edits and displays "nCount" data.
key="Id"	States that the "Id" field of the table's Product records will be used to uniquely identify each nCount entry.

The second step is to process the data that is entered into the table after the form is submitted. The following script illustrates this:

```
function update( msg )
{
// Retrieve a list of all parameters sent to the script
var list = msg.getFieldList();
for ( i = 0; i < list.getLength(); i++ )
{
// Retrieve the name of the next parameter
var strParam = list.getName( i );

// If the param came from a table "nCount" cell, get the value typed
in
var strCellValue = Table.getValue( "nCount", strParam, msg );

if ( strCellValue != "" )
{
// Get the key value of the cell
var strRowKey = Table.getKey( "nCount", strParam );
```

```
// Do something with strRowKey and strCellValue
...
}
}
```

This code reads all parameters sent to the script. It then determines those that are part of the `nCount` column in the table. Finally, it obtains the unique key that identifies the value. Once the script obtains these pieces of data it can do whatever is appropriate in the application.



Appendix A

Weblication Reference

This chapter is a reference for the weblication Extensible Markup Language Document Type Definition (XML DTD). XML DTD is the high level XML language used to define all Get.It! weblications.

Weblication Structure

All XML structure is comprised of tags with supporting attribute and element information. All basic weblications have the following structure:

```
<application>
  <modules>
    <module>
      <components>
        <component>
        </component>
      </components>
      <activities>
        <activity>
          <forms>
            <form>
            </form>
          </forms>
        </activity>
      </activities>
    </module>
  </modules>
</application>
```

Weblications are defined by an initial application entry or tag. The application tag is comprised of one or more modules (e.g., *service*, *request*, *approval*, *status*, and *receiving*). Modules contain elements called components. Each module contains one or more activities (e.g., the request module contains the following activities: browse catalog, review shopping cart, submit order, retrieve saved cart). Each activity can have one or more forms (e.g., the request browser catalog activity has the following forms: category menu, product list, product detail, bundle list, bundle detail).

Weblication Tags

<application>

The <application> element is the starting point for defining a weblication. It accepts the following attributes and nested elements:

Attribute	Description
<code>name</code>	A unique name for the weblication. The name should be a single word starting with a letter.
<code>onload</code>	The name of the script that will be invoked when the weblication's main menu is displayed. For example, the <code>login.xml</code> application file contains the onload <code>login.login</code> . The script name is made up of the script file name followed by the script function name.
<code>param</code>	When a script is defined with the onload attribute then it can define parameters that are included within the request message sent to the script. The string value should be constructed as <code>key=value</code> pairs with multiple parameters separated by ampersands (&). Because an ampersand is a special character in XML, you should use <code>&& (_amp)</code> . The plus sign (+) is converted into a space character. Special characters can be encoded as a three character string beginning with the percent sign followed by a two-digit hexadecimal representation of the lower 8-bits of the character. For example: <pre><application onload="login.login" param="p1=yada\$\$ (_amp)p2=what+ever">.</pre>
<code>home</code>	Designates the module which is to be considered the "Home" module and is reached by clicking on the first module tab. The label for the home tab will be the short name given to the module in the <code>description</code> attribute.
<code><title></code>	Title used in the application's main menu.
<code><instructions></code>	Instructions that will be displayed in the application's main menu.
<code><modules></code>	List of modules that make up the application.
<code>frame</code>	Specify whether you want the Get.It! banner to frame your windows. <code>frame="true"</code> causes the banners to display. <code>frame="false"</code> causes the banner to not display.

<module>

The <module> element defines an application component designed to offer users a specific application function. For instance, the Request module defines interfaces that permit users to create purchase requests. This element can contain the following attributes and nested elements:

Attribute	Description
<code>name</code>	A unique name for the module. The name should be a single word starting with a letter.
<code>access</code>	Defines the name assigned to a user-access definition that is required in order to access the module. User access is defined by capability words in ServiceCenter and UserRights in AssetCenter and is set for each user profile. See “User Authentication” on page 3-3 of the Get.It! Administrator’s Guide for more information. Enter a valid capability word or UserRight, or for more general access enter one of the following: anonymous = The module can be accessed by any user, regardless of the user’s profile capabilities. The module can even be accessed by users who are not logged into the Weblication. all = The module can be accessed by all users who are logged into Get.It!
<code>access-redirect</code>	The URL displayed when a user is denied access based on the “access” attribute (above). If no default URL is specified then <code>e_login_main_refuse.jsp</code> is used.
<code>appmenu</code>	Controls whether the module is included in the header shortcut menu. When set to <i>false</i> the module is not listed in the Weblication’s header shortcut menu. The default is <i>true</i> .
<code>apphead</code>	Controls whether the module is included on the main menu. When set to <i>false</i> the module is not listed in the weblication's main menu form. The default is <i>true</i> .
<code><title></code>	The title used to identify the module.
<code><description image="X" short="Y" long="Z"></code>	This element defines attributes that further describe the module. <ul style="list-style-type: none">• The image attribute defines an image that can be used as a module logo or link. This is a URL (relative or absolute) pointing to a specific image of browser-supported file type.• The short attribute should be defined by one or two words that can be used in a link that takes a user to the module.• The long attribute should contain a longer description.

Attribute	Description
<code><target URL></code>	Link a different module into this module. You can link any URL. See “Adding a URL as a Module” on page 4-23 of the Get.It! Tailoring Guide for details.
<code><components></code>	The list of forms and subforms that can be used in activities. These must be defined as components. See “Reusable Form Components (Subforms)” on page A-33 for details.
<code><activities></code>	List of activities that comprise the module.

Note: If you did not want this module included in the header menu, you would have included

`<appmenu="false">` before the `<title>` attribute.

<activity>

The `<activity>` element defines a step within a module's functionality. For instance, the **browse** activity in the Request module defines interfaces that allow users to browse the catalog to make a request.

This element can contain the following attributes and nested elements:

Attribute	Description
<code>name</code>	A unique name for the activity. The name should be a single word starting with a letter.
<code>access</code>	The name of a user <i>capability word</i> that is required to access the activity. The default value is anonymous , meaning that the activity may be accessed by any user, regardless of that user's profile capabilities. An anonymous activity can be accessed by users that are not even logged into the weblication.
<code><title></code>	Title used to identify the activity.

Attribute	Description
<pre><description image="X" short="Y" long="Z"></pre>	<p>This element defines attributes that further describe the activity.</p> <ul style="list-style-type: none"> • The image attribute defines an image that can be used as an activity logo or link. • The short attribute should be defined by one or two words that can be used in a link that takes a user to the activity. • The long attribute should contain a longer description that is used as balloon help for links to the activity.
<pre><target URL></pre>	<p>Link a module as an activity. You can link any URL. See “Adding a URL as an Activity” on page 4-24 for details.</p>
<pre><forms></pre>	<p>List of forms that comprise the activity.</p>

See figure A.1 for a sample of how the `<activity>` weblication tag can be used.

```
<activity name="catalog">
  <description short="Browse Catalog"
    long="Browse product and service catalog to make order selections"/>
```

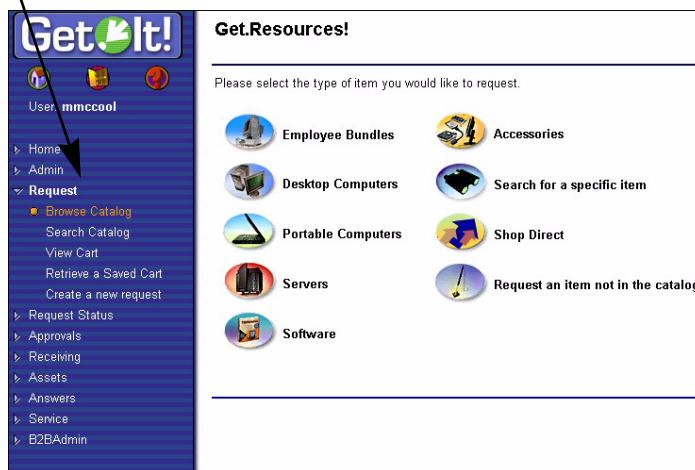


Fig. A.1 Using the `<activity>` tag

<form>

The specific contents for a screen are defined in the <form> element. For instance, the browser activity in the request module has a number of forms used to show product categories, product lists, product details, and so forth.

This element can contain the following attributes and nested elements:

Attribute	Description
<code>name</code>	A unique name for the form. The name should be a single word starting with a letter.
<code>onload</code>	Name of script to invoke before displaying the form. The message returned by the script is used to populate fields in the form.
<code>param</code>	When a script is defined with the <code>onload</code> attribute then it can define parameters that are included within the request message sent to the script. The string value should be constructed as <code>key=value</code> pairs with multiple parameters separated by ampersands (&). Because an ampersand is a special character in XML, you should use <code>&&(_amp)</code> . The plus sign (+) is converted into a space character. Special characters can be encoded as a three character string beginning with the percent sign followed by a two-digit hexadecimal representation of the lower 8-bits of the character. For example: <pre><application onload="login.login" param="p1=yada\$\$(_amp)p2=what+ever"></pre>
<code>onbrowserload</code>	Name of the script to invoke on the client before displaying the form.
<code>homepage</code>	If set to <code>true</code> , the form is created to become the webication's homepage. Only one form should be given this attribute.
<code><redirect></code>	This element defines a condition that is evaluated before displaying the form. If the condition is <code>true</code> , an alternative form is displayed instead. See “<redirect>” on page A-8 for more information.
<code><title></code> TEXT <code></title></code>	Title used to identify the form.
<code><instructions></code> TEXT <code></instructions></code>	Text giving the user instructions for the form.
<code><form fields></code>	One or more elements that make up the form, such as entry fields, labels, tables, menus, etc. See “form fields” on page A-10 for more information.
<code><actions></code>	Definition of actions that a user may take when viewing the form. These are typically displayed as buttons or links that submit the contents of the form or send the user to another form.

Attribute	Description
<code>type="frameset"</code>	Specifies number of frames. Also accepts <code>rows=""</code> and <code>cols=""</code> attributes.
<code>sidebar</code>	<code>Sidebar</code> refers to the leftmost frame. Setting this attribute to <code>false</code> eliminates this frame. <code><form name="xyz" sidebar="false"></code>
<code>frame</code>	<code>Frame</code> refers to the uppermost frame. Setting this attribute to <code>false</code> eliminates this frame. <code><form name="xyz" frame="false"></code>

Note: When a form is loaded to send to a client, it is supplied with an input document. The input document is a representation of an XML document containing the data to be displayed in the form. In most cases, the form's input document is obtained by executing the form's `onload` script. The script returns a message object which represents the document.

Another important point to understand is that a form is frequently invoked with a number of parameters. Normally these parameters are made up of the values entered in input fields within the previous form. These parameters are passed on to the form's `onload` script.

See figure A.2 for a sample of how the `<form>` tag can be used.

The screenshot shows a web application with a sidebar menu and a main content area titled "Search Results". The sidebar includes a "Request" section with options like "Browse Catalog", "Search Catalog", and "View Cart". The main content area displays a table of search results for IBM laptops. A callout box highlights the HTML code for the search results table, showing the use of the `<form>` tag and the `<redirect>` tag.

```

<form name="catalog" onload="procure.getCatalog">
  <redirect target-form="catalognone" condition="catalognone"/>
  <title> $$$(Title) </title>
  <instructions>
    Here are the items found in this category. You may click on any one to see a detailed description, or you may simply enter a count to add items to your request.
  </instructions>
  <table record="Product" rows="10">
    <link target-form="product" field="Id"/>
    <column label="Count" type="select" field="nCount" key="Id" record="Combo" valuelist="Value" displaylist="DisplayValue" />
    <column label="Brand" field="Brand"/>
    <column label="Model" field="Model"/>
    <column label="Price" field="Price"/>
  </table>
  <actions target-activity="review">
    <submit name="bTable"> Add to shopping cart </submit>
    <back/>
  </actions>
</form>

```

Fig. A.2 Using the `<form>` tag

<redirect>

This element defines a condition that is evaluated before displaying the form. If the condition is true, an alternative form is displayed instead.

For instance, a weblication could have the following:

```

<form name="hello" onload="weather.getTemperature">
  <redirect target-form="coats" condition="cold"/>
  <redirect target-form="shorts" condition="hot"/>
</form>

```

The code above would redirect the user to the `coats` page when `weather.getTemperature` returns a condition of `cold`.

It redirects to `shorts` when the condition is `hot`. It is the script's responsibility to establish a condition value that makes the redirection work. This is accomplished via the `Message.setCondition()` method.

The `<redirect>` element can take the following attributes:

Attribute	Description
<code>TARGET</code>	Defines the target location for the form.
<code>condition</code>	The condition value that makes the statement execute. If the condition value matches the value set in the form's script return message, the redirection will take place. If no condition is provided, the redirection is always executed.

In the previous example, we used the following string, which includes the `<redirect>` tag:

```
<form name="catalog" onload="procure.getCatalog">  
  <redirect target-form="catalognone" condition="catalognone"/>  
  <title> $$ (Title) </title>
```

When no catalog can be found (the condition of "catalognone") the following string of tags is used:

```
<!-- This form is shown when the search found nothing -->  
  <form name="catalognone">  
    <title>Search Results</title>  
    <instructions>  
      No catalog entries were found to match search criteria.  
    </instructions>  
    <actions>  
      <back/>  
      <home> Home </home>  
    </actions>  
  </form>
```

Fig. A.3 Using the `<redirect>` tag

component

A component is an element that can be used (and reused) within an activity's form. The component is defined separately and then referenced in the form by name. It may contain a number of fields or elements that are used to display and input data.

Attribute	Description
<code><name></code>	The unique name for the component used as reference within an activity's form using the tag <code><component name="<name>" /></code> .
<code>{form fields}</code>	One or more elements that comprise a form, such as entry fields, labels, tables, menus, and other (these elements are described in <code>{form fields}</code>).

form fields

A form may contain a number of fields or elements that are used to display and input data. Each is described in detail separately below. The following is the list of possible elements:

Attribute	Description
<code><fields></code>	Groups one or more "field" elements, which include <code><input></code> and <code><field></code> elements. Sample fields include text boxes, combos, check boxes, static text fields, and input fields. When fields are grouped they are treated as a group by the weblication, meaning the field labels are aligned and the input fields are aligned in the window automatically.
<code><menu></code>	A menu of links.
<code><table></code>	A table whose rows are obtained dynamically at run time from the form's input message.
<code><listbox></code>	A table whose rows are pre-defined within the weblication.
<code><html></code>	Allows the insertion of arbitrary HTML code.
<code><entry table></code>	A table that allows entries in one column and contains descriptions in another.
<code><plug in></code>	Allows you to plug in content from any web page that is accessible through a URL.

Search Results

Here are the items found in this category. You may click on any one to see a detailed

Count	Brand	Model
<input type="text"/>	IBM	THINKPAD 390E PII/300 PE 4.3GB 64MB
<input type="text"/>	IBM	THINKPAD 390E PII/333 PE 6.4GB 64MB
<input type="text"/>	IBM	THINKPAD 570 PII/333MHZ 4.0GB 64MB
<input type="text"/>	IBM	THINKPAD 570 PII/300MHZ PE 4.0GB

```

<form name="catalog" onload="procure.getCatalog">
  <redirect target-form="catalognone" condition="catalognone"/>
  <title> ${Title} </title>
  <instructions>
    Here are the items found in this category. You may click on any one to
    see a detailed description, or you may simply enter a count to add items
    to your request.
  </instructions>
  <table record="Product" rows="10">
    <link target-form="product" field="Id"/>
    <column label="Count" type="select" field="nCount" key="Id"
      record="Combo" valuelist="Value" displaylist="DisplayValue" />
    <column label="Brand" field="Brand"/>
    <column label="Model" field="Model"/>
    <column label="Price" field="Price"/>
  </table>
  <actions target-activity="review">
    <submit name="bTable"> Add to shopping cart </submit>
    <back/>
  </actions>
</form>

```

Fig. A.4 Using the table element of the form fields

<fieldtable>

A `<fieldtable>` element allows the creation of a formatted table of entry fields. For example, the Request form is displayed using `<entrytable>`. This element is used in the following manner:

```

<fieldtable>
  <heading> Section heading ... </heading>
  <row>
    <input> or <field>
    <input> or <field>
    ...
  </row>
  ...
</fieldtable>

```

The sample below shows the tag as it is used in the Request form definition in the request.xml file.

```
<component name="requestform">
  <fieldtable>
<heading> When would you like this and what is it for? </heading>
  <row>
<input label="Date"
type="date"
field="RequestedFor"
scope="user" />

<input label="Purpose"
type="text"
field="Purpose"
size="35"
scope="user"
required="true" />
  </row>

<heading> Who is this for and where should it be delivered?
</heading>
  <row>
<input label="First"
type="text"
field="FirstName"
scope="user"
required="true" />
<input label="Location"
type="text"
field="LocationName"
scope="user" size="35" />
  </row>
  <row>
<input label="Last"
type="text"
field="LastName"
scope="user"
required="true" />
<input label="Address"
type="text"
field="Address1"
scope="user"
size="35" />
<input type="hidden"
field="Address2"
scope="user"
value="$$ (Address2)" />
  </row>
  ...
</row>
```



```

<input label="Project"
type="select"
field="Project"
record="Project"
valuelist="Title"
displaylist="Title"
scope="user" />
<input type="textarea"
field="Comment"
rows="3"
cols="35"
scope="user"
colspan="2"
rowspan="2"/>
</row>
<row>
<input label="Budget"
type="select"
field="Budget"
record="Budget"
valuelist="Name"
displaylist="Name"
scope="user" />
</row>

<heading> Request contents: </heading>
</fieldtable>

```

This code, when displayed in Get.It!, is shown below.

The screenshot shows the 'Submit New Request' page in the Get.It! application. On the left is a navigation menu with options like 'Home', 'Admin', 'Request', 'Request Status', 'Approvals', 'Receiving', 'Assets', 'Answers', 'Service', and 'E2BAdmin'. The main content area contains the following sections:

- When would you like this and what is it for?:** Includes a date picker (Sep 18 2000) and a purpose dropdown (Another computer).
- Who is this for and where should it be delivered?:** Includes fields for First Name (Matthew), Last Name (Mccool), Location, Address, City, State, and Zip Code.
- What department is paying for this?:** Includes fields for Cost Center, Project, Budget, and Attachments (status.doc).
- Select individual request lines to edit details:** A table with columns: Count, Brand, Model, Description. One row is visible: 1, IBM, THINKPAD 570 PII300MHZ PE 4.0GB.
- Grand Total:** \$2,699.00
- Buttons:** 'Submit for Approval' and 'Save for Later'.

Fig. A.5 The <fieldtable> tag in use.

The following attributes can be specified within the <input> or <field> elements in a

row:

Attribute	Description
<code>colspan=N</code>	Normally an input field fills out two columns in a table: a column for its label, and a column for the field. However, you can use <code>colspan</code> to specify that the field should take up both columns. For example: <pre data-bbox="565 527 1414 554"><input type="textarea" field="description" colspan="2" ...></pre> The field above is given no label and is defined to span two columns. Therefore, the text area takes up both the label and entry columns in a table. Typical values for <code>colspan</code> are 2 or 4.
<code>rowspan=N</code>	Allows a field to span more than one row in height. This is also typically used with <code>textarea</code> fields in a <code>fieldtable</code> .

<action>

The `<action>` element contains actions that a user may take when viewing the form. These are typically displayed as buttons or links that submit the contents of the form or send the user to another form.

The element may contain several attributes and nested elements. Consider the following example which is referenced by the descriptions of these attributes and elements below:

```
<actions target-activity="review">  
  <submit> Add to shopping cart" </submit>  
  <submit name="Remove"> Remove from cart </submit>  
  <link target-form="help"> Help </link>  
  <back/>  
</actions>
```

Attribute	Description
<code>TARGET</code>	<p>Defines the destination where the user is taken when the current form is submitted. Currently, each form may only have one submit destination. In the sample above, the <code>TARGET</code> for the actions is the <i>review</i> activity of the current module.</p>
<code><submit></code>	<p>Defines a submit button for a form. In the example above, the first submit entry displays a button with the caption <i>Add to shopping cart</i>. Clicking the button sends you to the form's action target (the <i>review</i> activity). Any data entered in the form is sent along to the target form and will be available to the target form's <code>onload</code> script.</p> <p>Forms typically have one submit button. However, forms with more than one submit button can differentiate between them using the optional name attribute.</p> <p>For example, notice the second submit button. It also sends the user to the form's target destination (the <i>review</i> activity). However, the script of the target form can distinguish that it was invoked with the <i>Remove from cart</i> button because the button's name is sent along with the form. The script can check for this as follows:</p> <pre>if (msg.get("Remove") != "") // form called with the "Remove" button ...</pre>
<code><link></code>	<p>Link actions are typically displayed by the weblication just like any submit button. However, a link button offers a way to send the user to any arbitrary <code>TARGET</code> destination. However, when a link is used, the form's data is not submitted to the target.</p>
<code><back></code>	<p>Creates a button that takes the user to the previous form.</p>
<code><home></code>	<p>Creates a button that takes the user to the home menu.</p>

The screenshot shows a web application interface with a sidebar menu on the left and a main content area. The sidebar menu includes items like Home, Admin, Request, Browse Catalog, Search Catalog, View Cart, etc. The main content area displays search results for IBM products. A callout box highlights the HTML code for the search results table, showing the use of the <action> tag for the 'Add to shopping cart' button.

```

<form name="catalog" onload="procure.getCatalog">
<redirect target-form="catalognone" condition="catalognone"/>
<title> $$ (Title) </title>
<instructions>
  Here are the items found in this category. You may click on any one to
  see a detailed description, or you may simply enter a count to add items
  to your request.
</instructions>
<table record="Product" rows="10">
  <link target-form="product" field="Id"/>
  <column label="Count" type="select" field="nCount" key="Id"
    record="Combo" valuelist="Value" displaylist="DisplayValue" />
  <column label="Brand" field="Brand"/>
  <column label="Model" field="Model"/>
  <column label="Price" field="Price"/>
</table>
<actions target-activity="review">
  <submit name="bTable"> Add to shopping cart </submit>
  <back/>
</actions>
</form>

```

Fig. A.6 Using the <action> tag

TARGET

Various weblication elements support a set of **TARGET** attributes that are translated into links to a browser destination. One of the powerful concepts in a weblication is its ability to make navigation between pages easy without requiring the developer to hard code actual destination page names.

The goal behind the target's design is to encapsulate the contents of each module and activity, reducing interdependencies. Therefore, the targets listed in the following table allow a developer to say something like "take me from the current activity to some other activity in this module." This is done without specifically listing the target form name, thus reducing dependencies which would make a weblication harder to maintain as modules and activities are added or rearranged.

The following are possible `TARGET` attributes:

Attribute	Description
<code>target-form</code>	Leads to a named form. This target is used for navigation within the current activity. That is, the target form must be in the current activity.
<code>target-activity</code>	Leads to the first form of the named activity. This target is used for navigation within activities of the current module. That is, the target activity must be in the current module.
<code>target-module</code>	Links to the first form of the first activity in the named module.
<code>target-url</code>	Links to any URL. Anything that could be used in an HTTP <code>href</code> tag can appear here.
<code>target-script</code>	Executes a client-side script when the button is pushed.
<code>target-field</code>	<p>Sometimes the target is not known until run time. This attribute causes the weblication to look for an input document field that contains a target URL. For example:</p> <pre><link target-field="VendorURL"> More information </link></pre> <p>The target above is evaluated at run time by retrieving the VendorURL from the form's input document.</p>
<code>param</code>	<p>This attribute can accompany any of the target attributes mentioned above. It defines additional parameters that should be sent to the target form. For example:</p> <pre><link target-form="catalog" param="Certification=Desktop"> Desktop Computers </link></pre> <p>The link above passes a parameter named Certification with a value of <i>Desktop</i> to the target catalog form.</p>

TEXT

Various Weblication elements support the display of arbitrary text. For example, form instructions are specified by the `<instructions>` element with some embedded text:

```
<instructions>Press button with mouse</instructions>
```

However, wherever an element is documented to support `TEXT`, you can enter more than just plain words. The text can contain embedded HTML mark-up elements, and it may also contain references to values in the form's input document. For instance:

```
<instructions>Press button with mouse. <br/>
```

```
If nothing happens <b>repeat until it works!!</b>
</instructions>
```

The instructions above have embedded HTML tags `
`, ``, and ``. Embedded HTML must be XML compliant. This means that each starting HTML tag should have an ending tag (e.g., ` ... `) or use the XML shorthand for the tag (e.g., `
` rather than `
`). Attributes inside HTML elements also must be quoted (e.g., `` rather than ``).

In addition, you can embed field values in text. For example:

```
<instructions>Hello $$(<UserName>),
How are you doing?
</instructions>
```

The `$(X)` syntax is used to extract a field from the form's input document.

\$(X)

The `$(X)` element is used to extract information from a field in the form's input document. It embeds field values in text. For example:

```
<instructions>Hello $$(<UserName>),
How are you doing?
</instructions>
```

This example will display the value in the `UserName` field within the form instructions.

Within the HTML contents, you can use `$(X)` expressions to include values of fields in the form's input document.

<menu>

The `<menu>` element creates a menu of links in a form. For example, the request module uses a menu to show catalog categories. The `<menu>` element can accept a series of `<links>`. This is supported if a `record` attribute is not specified for the menu. When a `record` attribute is specified it behaves similar to the `<table>` element. This means that it accepts a `rows=10` attribute so that when there are more than ten items you can insert both Next and Previous buttons on the bottom of the page.

```
<menu record="{MessageElement}" rows="{max-rows-displayed}"
  image="{field-with-image-path}"
  label="{field-with-display-string}">

<link target-form="{target-form}" field="{field-with-ID-key}"
  content="{optional-context-value}"/>
</menu>
```

OR

```
<menu record="Category" rows="10" image="ImageName" label="Name">  
<link target-form="subcategory" field="Id"/>  
</menu>
```

If this is implemented as a table then it will appear as in the following.

```
<table record="Category" rows="10">  
<link target--form="subcategory" field="Id"/>  
<column field="ImageName" type="image"/>  
<column field="Name"/>  
</table>
```

Attribute	Description
<code><link></code>	Defines an item in the menu. Each <code><menu></code> tag should have one or more embedded <code><link></code> tags. Link attributes are described in the table below.

Link Attributes

Attributes	Description
<code><link TARGET></code>	Defines the destination target of the link.
<code><link image=X></code>	An image URL to use for the menu link.
<code><link window="true"></code>	If this attribute is set, the target of the link is displayed in a separate browser window.
<code><link></code> <i>TEXT</i> <code></link></code>	The text used in the link.

Note: `<link>` elements may also appear inside a `<fields>` collection.

<table>

The `<table>` element provides a concise way to create tables in the form. This tag is specialized in generating tables that are populated with XML documents obtained from database queries. The following attributes and embedded elements are supported:

Attribute	Description
<code>record</code>	<p>This attribute identifies the specific record the table is designed to display. This record type is found in the form's input XML document. For instance, consider the following document:</p> <pre><recordlist> <Product> <Brand> X </Brand> <Price> 1 </Price> <ProductId> 1356 </ProductId> <nCount> 1 </nCount> </Product> ... </recordlist></pre> <p>To display a table with a list of products, the <code>record</code> attribute is set to <code>Product</code>. (This sample XML is used in the examples below.)</p>
<code>rows</code>	<p>The max number of rows to display in the table. If the query result set for the table is larger than this number of rows, the table automatically displays "Next" and "Previous". If this attribute is not specified, the table is made as large as needed to display all rows in the record set.</p>
<code><link TARGET field=X></code>	<p>Table element used to make the rows in a table into links to another form. For example, a catalog table has rows that when clicked display each product's detail. This element takes two attributes. The <code>TARGET</code> attribute determines where the link is to take the user. The <code>field</code> attribute is used as a parameter passed to the target page. It is intended to uniquely identify the row. For example:</p> <pre><link target-form="details" field="ProductId" target="detail"/></pre> <p>The link above comes from the <code>catalog</code> table. It creates row links that take the user to the <code>details</code> form. In addition, the <code>ProductId</code> field of the row's record is sent along as a parameter to the target form. This way, the target form can be initialized to display the correct details. The <code>detail</code> field is for inserting a URL.</p>

Attribute	Description
<code><column></code>	Each table should have one or more columns. Columns can be used to display a variety of things, including static text, pictures, and entry fields. Each type of column is described in more detail below:
<code><submit></code>	Allows the transmission of data upon pressing the relevant button. When set to true the form's contents are submitted to the target of the link. User globals are updated. Other changes may be saved.
<code><submit target-X></code>	This supplies a target-X in a <code><submit></code> element. For instance, <pre> <actions target-form="someDefault" <submit>Save</submit> <submit target-activity="review"> Review Before Saving</submit> </actions> </pre>

Column Types

Type	Description
Static Text	The default content of a column is static text. The <code>label</code> attribute specifies the column's heading. The <code>field</code> attribute defines the record field to display in the column. For example, <pre> <column label=X field=X> </pre>
Entry Field	These columns display a text <i>entry</i> field where the user can type in some text. The <code>label</code> and <code>field</code> attributes serve the same purpose as those of static text columns. The <code>key</code> attribute should contain the name of a record field that uniquely identifies each row in the column. <pre> <column type="entry" label=X field=X key=X size=X> </pre> <p>The optional <code>size</code> attribute defines how wide to make the entry fields (in number of characters). For example:</p> <pre> <column label="Count" field="nCount" type="entry" key="ProductId" size="3"/> </pre> <p>This is a column in the product catalog table that lets users enter a count with the number of products to order. The column displays the <code>nCount</code> field from the table's record. The column uses each row's <code>ProductId</code> to uniquely identify the entry fields. This is necessary so that scripts that interpret the input entered in a table can match up table entries with an application or item context.</p>

Type	Description
<p>Select Box (populated dynamically)</p>	<pre data-bbox="776 365 1382 422"><column type="select" label=X field=X key=X record=X valuelist=X displaylist=X ></pre> <p data-bbox="776 432 1459 625">You can display a select box or combo-box in a column with a list of valid entry choices from which the user can choose. The choices are obtained dynamically from the form's input document. The attributes listed here work the same way as described for entry field columns. There are two additional attributes: <code>valuelist</code> and <code>displaylist</code>. These are used to specify the name of the record field containing the choices for the select box. For example:</p> <pre data-bbox="537 653 1297 737"><column label="Project" type="select" field="ProductProject" record="Project" valuelist="Id" displaylist="Title"/></pre> <p data-bbox="776 758 1463 835">This column displays select boxes with a list of <code>Project</code> choices. For this to work, the form's input document should include <code>Project</code> entries such as:</p> <pre data-bbox="776 873 1338 1073"><recordlist> <Project> <Id> 123 </Id> <Title> New Development 99 </Title> </Project> ... </recordlist></pre> <p data-bbox="776 1087 1463 1192">The selected choice is associated with the <code>ProductProject</code> field of the table's <code>Product</code> record. The choices displayed are determined by the <code>Title</code> field on the <code>Project</code> records, and the actual values submitted for each choice are those of the <code>Id</code> field in the <code>Project</code> records.</p>
<p>Select Box (populated statically)</p>	<pre data-bbox="776 1220 1409 1247"><column type="select" label=X field=X key=X></pre> <p data-bbox="776 1262 1463 1339">Columns can display select boxes with statically defined choices. The label, field, and key attributes are the same as those defined above. Here is an example:</p> <pre data-bbox="537 1367 1328 1478"><column label="Approval" type="select" field="Approve"> <option value="1"> Yes </choice> <option value="0"> No </choice> </column></pre> <p data-bbox="776 1503 1317 1530">This column displays <code>Approval</code> choices of <i>Yes</i> and <i>No</i>.</p>
<p>Lookup</p>	<p data-bbox="776 1556 1263 1583">Opens a searchable, pop-up window.</p>
<p>Image</p>	<pre data-bbox="776 1608 1122 1635"><column label=X field=X></pre> <p data-bbox="776 1646 1458 1696">This column displays an image. The image's URL is obtained from the specified field in the table's input record.</p>

Type	Description
Radio Button	<pre data-bbox="634 365 1317 422"><column label="Current employee" type="radio" field="Field1" /></pre> <p data-bbox="634 432 1317 516">The <code>field</code> attribute specifies the record field in the form's input document that should be used to populate the field's value. See “<columns>” on page A-23 for more information.</p>
<pre data-bbox="394 541 605 598"><link TARGET field=X></pre>	<p data-bbox="634 541 1317 705">Table element used to make the rows in a table into links to another form. For example, a catalog table has rows that when clicked display each product's detail. This element takes two attributes. The <code>TARGET</code> attribute determines where the link is to take the user. The <code>field</code> attribute is used as a parameter passed to the target page. It is intended to uniquely identify the row.</p>
Label	The column label.
Image	This is a static image file name, which can be provided as an alternative to “field”
Balloon	Used only when target-X is defined. This specifies balloon help for the image link

<columns>

It is possible to split a weblication form into columns, as shown in the following:

```
<columns>
  <column>
    Weblication elements for this column ....
  </column>
  <column>
    Weblication elements for this column
  </column>
</columns>
```

The sample below shows the tag as it is used in the `request.xml` catalog category window.

```
<columns>
  <column>
    <fields>
      <link target-form="bundles" image="images/catbundle.gif">
        Employee Bundles </link>
      <link target-form="catalog" param="Certification=Desktop"
        image="images/catdesktop.gif"> Desktop Computers </link>
      <link target-form="catalog" param="Certification=Laptop"
        image="images/catportable.gif"> Portable Computers </link>
    </fields>
  </column>
</columns>
```

```

        <link target-form="catalog" param="Certification=Server"
            image="images/catserver.gif"> Servers </link>
        <link target-form="catalog" param="Certification=Software"
            image="images/catsoftware.gif"> Software </link>
    </fields>
</column>
<column>
    <fields>
        <link target-url="e_b2bshop_return_b2blist.jsp"
            param="ListAction=B2BShopOnly"
            image="images/catshopdirect.gif"> ShopDirect </link>
        <link target-form="catalog"param="Certification=Accessories"
            image="images/cataccessories.gif"> Accessories </link>
        <link target-activity="offcatalog"
            image="images/catoffcat.gif"> Request an item not in the
            catalog </link>
        <link target-form="search" image="images/catsearch.gif">
            Search for a specific item </link>
    </fields>
</column>
</columns>

```

This code, when displayed in Get.It!, is shown below.

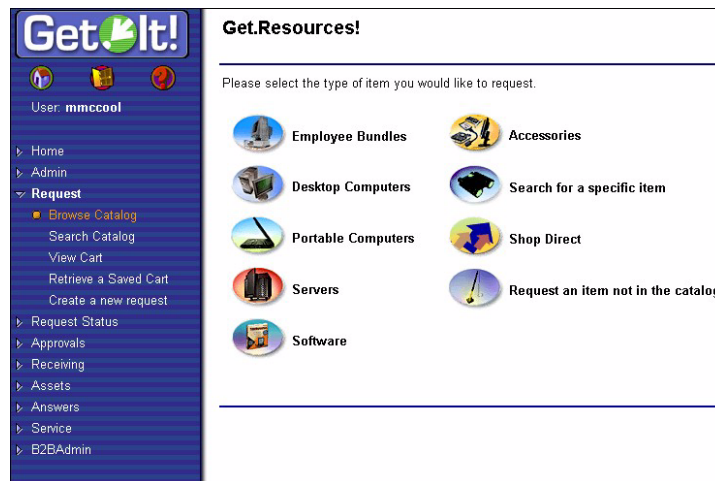


Fig. A.7 The <column> tag in use.

<listbox>

The <listbox> element is used to display a table in a form. However, unlike the <table> element, listbox tables contain rows that are statically defined in the weblica-

tion. For example, a listbox is used to display the details of a Knowledge solution in the service module. (See the *solution* form in the `service.xml` file).

The following attributes and nested elements are supported:

Attribute	Description
<code><heading></code>	Defines the listbox headings. This element should be followed by one or more nested <code><field></code> elements that describe each heading.
<code><row></code>	Defines a row in a listbox. This should be followed by one or more nested <code><field></code> elements that are part of the row.
<code><field image=X field=X> Text </field></code>	A single element that may be placed in a heading or row cell. The optional <code>image</code> attribute may point to an image URL to display for the field. The <code>field</code> attribute may point to a field from the form's input document. Otherwise, the field displays its text contents.
<code><input></code>	You can enter any valid input element. See “ <code><input></code> ” on page A-26 for types of input elements.

Here is a sample listbox that results in a small table with phone numbers to call to contact support or sales.

```
<listbox>
  <heading>
    <field> Name </field>
    <field> Phone </field>
  <heading>
  <row>
    <row>
      <field> Customer Support </field>
      <field> 123-4567 </field>
    </row>
    <field> Sales </field>
    <field> 765-4321 </field>
  </row>
</listbox>
```

<field>

The `<field>` element creates a static text or image field on a form. These elements must

be placed within the `<fields>` parent element. The following attributes are supported:

Attribute	Description
<code>label</code>	Specifies the label for the field.
<code>field</code>	Used to determine the value of the field. This attribute specifies the record field in the form's input document that should be used to populate the value text field.
<code>type</code>	Specifies the type of data expected for this field. The default is text type data. If the type is set to <code>image</code> , the field's value is assumed to be a URL to an image.
<code><link TARGET field=X></code>	Table element used to make the rows in a table into links to another form. For example, a catalog table has rows that when clicked display each product's detail. This element takes two attributes. The <code>TARGET</code> attribute determines where the link is to take the user. The <code>field</code> attribute is used as a parameter passed to the target page. It is intended to uniquely identify the row. For example:
<code><submit></code>	Allows the transmission of data upon pressing the relevant button. When set to <code>true</code> the form's contents are submitted to the target of the link. User globals are updated. Other changes may be saved.
<code><submit target-X></code>	This supplies a target-X in a <code><submit></code> element. For instance, <pre><actions target-form="someDefault" <submit>Save</submit> <submit target-activity="review">Review Before Saving</submit> </actions></pre>
<code><field>TEXT<field></code>	The value displayed in the field, displayed if no <code>field</code> attribute is already defined.

Using the `<submit>` element will depend upon your data transmission destination. When set to true, the form's contents are submitted to the target of the link. The `<submit>` tag sends data to the default destination. The `<submit target-x>` tag sends data to a defined destination.

`<input>`

The `<input>` element is used to create a variety of entry fields. Each type of field is

described in its own section. Here we define a list of attributes shared by all input fields:

Attribute	Description
<code>label</code>	Specifies the label for the input field.
<code>field</code>	Used to determine the value of the field. This attribute specifies the record field in the form's input document that should be used to populate the field's value.
<code>type</code>	Specifies the type of field. Examples include <code>submit</code> , <code>spinner</code> , <code>password</code> , and <code>composite</code> . Additional examples can be found in the <code>... \apps \service \weblications \servicelookup.xml</code> file.
<code>class</code>	Denotes the value of <code>input</code> , such as Action Button.
<code>value</code>	Normally the value is taken from the <code>field</code> attribute to extract a field value from the form's input document. However, if a value is specified explicitly, it will be used when displaying the form.
<code>scope</code>	Normally data entered in fields is sent along to the server and then forgotten. However, fields can be given a longer term scope, making their values available beyond a single submit. Right now, only one scope is supported: <code>scope="user"</code> . When set, the values entered in a field are stored in the current user session scope. When the form is displayed again, or when other forms display <code><input></code> elements for a field with user scope, the last value entered is always remembered.
<code>required</code>	If <code>true</code> , the field is flagged as being required. The form will not be submitted unless the user provides data for the field.

`<input>` (Text Field)

The `<input>` element is used to create a variety of entry fields. Below are the attributes used to define a single line entry field.

Attributes	Description
<code>label</code>	Specifies the label for the input field.
<code>field</code>	Used to determine the value of the field. This attribute specifies the record field in the form's input document that should be used to populate the field's value.
<code>type="text"</code>	To create text entry fields, type should be set to "text". This is the default value.

Attributes	Description
<code>value</code>	Normally the value is taken from the <code>field</code> attribute to extract a field value from the form's input document. However, if a value is specified explicitly, it will be used when displaying the form.
<code>size</code>	Defines the width of the text entry field in characters.
<code>readonly="true"</code>	Removes ability to edit. If this attribute is set, the text area will not be editable. This works only with the Internet Explorer browser.

<input> (Text Area)

The `<input>` element is used to create a variety of entry fields. Below are the attributes used to define a multiline entry text area.

Attribute	Description
<code>label</code>	Specifies the label for the input field.
<code>field</code>	Used to determine the value of the field. This attribute specifies the record field in the form's input document that should be used to populate the field's value.
<code>type="textarea"</code>	To create multiline text entry fields, type should be set to "textarea".
<code>value</code>	Normally the value is taken from the <code>field</code> attribute to extract a field value from the form's input document. However, if a value is specified explicitly, it will be used when displaying the form.
<code>rows</code>	Number of rows in the textarea.
<code>cols</code>	Width of the textarea in number of characters.
<code>readonly="true"</code>	Removes ability to edit. If this attribute is set, the text area will not be editable. This works only with the Internet Explorer browser.

<input> (Combo/Selection Box)

The `<input>` element is used to create a variety of entry fields. Below are the attributes

used to define a select box.

Attribute	Description
<code>label</code>	Specifies the label for the select box.
<code>field</code>	Used to determine the value of the field. This attribute specifies the record field in the form's input document that should be used to populate the field's value.
<code>type="select"</code>	To create text entry fields, type should be set to "text". This is the default value.
<code>record</code>	Specifies the record in the form's input document that contains the list of display and value lists.
<code>valuelist</code>	Specifies the field in the select box record that contains the values for each of the select choices.
<code>displaylist</code>	Specifies the field in the select box record that contains the labels for each of the select choices.

For example, consider:

```
<input label="Budget" type="select" field="RequestBudget"
record="Budget" valuelist="BudgetId" displaylist="Name" />
```

This generates a combo box with a label of *Budget*. The choices in the combo box are populated by looking at records of type *Budget*. The current selection is obtained from the `RequestField` field in the form's input document.

You can also define selection boxes with static choices (instead of populating the choices from a database record). Here is a sample:

```
<input type="select" label="Approval" field="Approve">
  <option value="1"> Yes </option>
  <option value="0"> No </option>
</input>
```

<input> (Checkbox)

The `<input>` element is used to create a variety of entry fields. Below are the attributes

used to define a checkbox:

Attribute	Description
<code>label</code>	Specifies the label for the checkbox.
<code>field</code>	Used to determine the value of the field. This attribute specifies the record field in the form's input document that should be used to populate the field's value.
<code>type="checkbox"</code>	To create checkboxes text entry fields, type should be set to "checkbox".
<code>value</code>	Specifies the value that the checkbox field should have when the checkbox is selected.
<code>valueoff</code>	When this attribute is specified two radio buttons are generated. The first displays the value (" <code>value=<text></code> ") when the check box is selected. The second displays the value when the checkbox is not selected. This form is also used for selecting one of two items. When <code>valueoff</code> is used the checkbox description is ignored.
<code><checkbox></code> Text <code></checkbox></code>	The checkbox description.

For example:

```
<input type="checkbox" label="Remember me" field="remember" value="true"> Enable automatic login </input>
```

This generates a checkbox associated with the form's *remember* field. If *remember* is set to *true* upon building the form, the checkbox will appear selected. If the user selects the checkbox the *remember* field is posted as *true* with the form.

<input> (Radio)

The `<input>` element is used to create a variety of entry fields. Below are the attributes

used to define a radio button:

Attribute	Description
<code>label</code>	Specifies the label for the radio button.
<code>field</code>	Used to determine the value of the field. This attribute specifies the record field in the form's input document that should be used to populate the field's value.
<code>type="radio"</code>	To create radio buttons, type should be set to "radio".
<code>value</code>	Specifies the value that the radio button should have when the radio is selected.
<code><radio></code> <code>Text</code> <code></radio></code>	The radio button description.

For example:

```
<input type="radio" label="Remember me" field="remember"
value="true"> Enable automatic login </input>
```

This generates a radio button associated with the form's `remember` field. If `remember` is set to `true` upon building the form, the radio button will appear selected. If the user selects the radio button the `remember` field is posted as `true` with the form.

<input> (Hidden)

Sometimes it is useful to create a hidden field in a form whose only purpose is to add some data that should be posted when the form's contents are sent back to the server. Below are the attributes used to define such a hidden field.

Attribute	Description
<code>field</code>	Used to determine the value of the field. This attribute specifies the record field in the form's input document that should be used to populate the field's value.
<code>type="hidden"</code>	To create text entry fields, type should be set to <code>text</code> . This is the default value.
<code>value</code>	Normally the value is taken from the <code>field</code> attribute to extract a field value from the form's input document. However, if a value is specified explicitly, it will be used when displaying the form.

<input> (Date)

The `<input>` element can be used to create a Date entry field. The field contains drop-down lists for day, month, and year. A calendar button can be pressed to display a calendar that can be used to select a specific day. Below are attributes to define a single line entry field.

Attribute	Description
<code>label</code>	This optional attribute specifies the label for the <code><input></code> field.
<code>field</code>	Used to determine the value of the field. This attribute specifies the record field in the form's input document that should be used to populate the field's value.
<code>type="date"</code>	To create date entry fields.
<code>value</code>	This is an optional attribute. Normally the value is obtained from the <code>field</code> attribute to extract a field value from the form's input document.
<code>startyear</code>	Number representing the last year in the drop-down list. This value can be a positive or negative number, whereby the given number is added to or subtracted from the current year. If this is omitted the default value will be 20 years prior to the current year.
<code>endyear</code>	Number representing the last year in the drop-down list. This value can be a positive or negative number, whereby the given number is added to or subtracted from the current year. If this is omitted the default value will be 20 years prior to the current year.

<input> (Money)

The `<input>` element can be used to create a Money entry field. The field contains drop-down lists displaying currency names as defined in the Currency property of the `archway.ini` file..

Attribute	Description
<code>label</code>	This optional attribute specifies the label for the input field.

Attribute	Description
<code>field</code>	Used to determine the value of the field. This attribute specifies the record field in the form's input document that should be used to populate the field's value.
<code>type="money"</code>	To create money entry fields. Specifies the type of data expected for this field. The default is "text" type data. If the type is set to "image," the field's value is assumed to be a URL to an image. If the type is "date" or "money," the value of the field is formatted to user's preferred locale.
<code>value</code>	Normally the value is taken from the field attribute to extract a field value from the form's input document. However, if a value is specified explicitly, it will be used when displaying the form.

<link>

The <link> element creates a hyperlink field in a form. For example, the request module uses a menu to show catalog categories. The following attributes and tags are supported in a menu:

Attribute	Description
<code><link TARGET></code>	Defines the destination target of the link. See "" on page A-16 for details about the TARGET attribute.
<code><link image=X></code>	An image URL to use for the menu link.
<code><link window="true"></code>	If this attribute is set, the target of the link is displayed in a separate browser window.
<code><link> TEXT </link></code>	The text used in the link.
<code><link field></code>	

Reusable Form Components (Subforms)

It is common for a webapplication to have several forms that need to display a common set of components. For example, in Get.Resources!, several forms display a detailed description of a request containing the request purpose, description, budget, department. These details appear in places like approval screens, request status screens, and shopping cart review

screens.

To address this need weblications support the definition of reusable component blocks, or subforms, that can be included wherever necessary. Reusable components are defined at the beginning of a module definition as shown in this example:

```
<module name="login">
  ...

  <components>
    <!-- Basic "login" screen -->
    <component name="login">
      <fields>
        <input type="text" label="User Name" field="loginuser"
          record="Employee" valuelist="Name" displaylist="Name"
          required="true"/>
        <input type="password" label="Password"
          field="loginpass"/>
        <break/>
        <input type="checkbox" label="Remember me"
          field="remember" value="true"> Enable automatic login
        </input>
      </fields>
    </component>
    ...
  </components>
```

This example defines a reusable subform named `login`. This block can then be inserted in any form as shown below:

```
<!-- This form lets the user logon -->
<form name="start" onload="login.init">
  <title> Welcome </title>
  <instructions>
    Please enter your user name and password to enter the
    Get,It! site
  </instructions>
  <component name="login"/>
  <actions target-url="appmenu.jsp">
    <login> Login </login>
    <link target-activity="register"> Register </link>
  </actions>
</form>
```

The contents of a `<component>` definition can be anything that is a valid form component, including tables, listboxes, and fields, etc. Forms can use any number of embedded component blocks, and they may include other form components as well.

Note: Components referenced in a form must be declared in the form's module. This makes most blocks reusable across all forms in a module. To define components that can be reused across modules, you should define the components in their own

files and use `<import>` statements to add them at the top of a `<module>` definition.

A component definition can include an `onload` attribute. This optional attribute names a script that should be invoked to provide data used by the component code. If this is provided, the document returned by the onload script is used for fields and `$$ (X)` expressions in the component instead of using the form's input document.

Additional Tags

`<html>`

The `<html>` tag allows the insertion of any arbitrary HTML code. This should be used with care, and only when the use of existing Weblication components is not sufficient. Within the HTML contents, you can use `$$ (X)` expressions to include values of fields in the form's input document.

The following attributes are supported:

Attribute	Description
<code>onload</code>	Names a script that should be invoked to provide data used by the HTML code. If this is provided, the document returned by the onload script is used in <code>\$\$ (X)</code> expressions instead of using the form's input document.
<code>param</code>	When a script is defined with the onload attribute then it can define parameters that are included within the request message sent to the script. The string value should be constructed as <code>key=value</code> pairs with multiple parameters separated by ampersands (&). Because an ampersand is a special character in XML, you should use <code>& (_amp)</code> . The plus sign (+) is converted into a space character. Special characters can be encoded as a three character string beginning with the percent sign followed by a two-digit hexadecimal representation of the lower 8-bits of the character. For example: <pre><application onload="login.login" param="p1=yada\$\$ (_amp)p2=what+ever"></pre>

Additional Functionality

Searchable Popup Window

Searchable Popup Windows (Lookup Window) are used for searching entries in a form. Implementing the Lookup field allows you to reuse it throughout the weblication. To use this function you must add a `lookup` input field to any form. The following example illustrates an Employee Lookup:

```
<input label="User"
  field="UserId"
  displayfield="UserFullName"
  type="lookup"
  readonly="true"
  balloon="Change User"
  target-module="commonlookup"
  target-activity="employee"/>
```

The `label` is the label or name of the field. `Field` is the hidden field that is to be returned to the server. The `displayfield` field is an optional display field name. The `type` field is the lookup type. The `readonly` field controls the edit/read function. The `balloon` field is for lookup help when the cursor is placed over an icon. The `target-module` field is for generic searching. The `target-activity` field is for searching activity.

Defining New Popup Windows

The first step to defining a new searchable popup window is to add an activity to `commonlookup.xml`, unless the lookup is module-specific whereby the activity can be added elsewhere. Searching activities take two forms. The first is to display search results. The second is to process the results. The following is an example of an Employee search:

```
<activity name="employee" menu="false">
  <forms>

    <!-- Basic search and result screen-->
    <form name="start" onload="lookup.getEmployees"
      frame="false" sidebar="false">
      <table record="Employee" rows="10">
        <link target-form="process" field="Id"/>
        <column label="Last" field="Name"/>
        <column label="First" field="FirstName"/>
      </table>
    </form>
```

Once the `lookup.getEmployees` function is in the code the `getX` and `selectX` scripts must be implemented. For generic activities this occurs in `common/jscript/lookup.js`. The following is an example of this script for the Employee search:

```
function getEmployees( msg )
{
  return archway.sendDocQuery( "ac", "Employee", msg );
}
```



The `selectX()` script processes and returns both the selection field ID and the selection display field.

```
function selectEmployee( msg)
{
  var msgResponse = new Message();
  var strId = msg.get( "Id" );

  // Return selected ID as the searchable field
  var strField = user.get ( "_lookupfield" );
  msgResponse.set( strField, strId );

  // Return employee's Full Name as the display field
  var strDisplayField = user.get ( "_lookupdisplay" );
  var msgEmp = archway.sendDocQuery( "ac",
    "SELECT FirstName,LastName,FullName FROM Profile WHERE Id=" =
    strId, 0, -1 );
  msgResponse.add( strDisplayField, msgEmp.get( "FullName" ) );
}
```

This script returns the selected Employee ID as the Lookup Field and then returns the Employee Full Name field as the LookupDisplay field. These values update all fields in the calling browser window. The lookup field is a hidden field returned to the server when a form is submitted. The Lookup Display field is a text or entry field to display a user-friendly version of the selection.



Appendix B

Document Schema DTD

This chapter is a specification reference for defining document schemas. See Chapter 3, "Introduction to Document Schemas," for additional information, including background and a complete example of a schema.

This chapter addresses:

- The Document Schema file template
- Schema attribute tags
- ServiceCenter-specific attributes

Document Schema Files

Define each document in its own schema file. The name of the schema file must match the document's name. For example, the Problem document is defined in [Problem.xml](#).

The structure of a schema file must fit the following template:

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

//filename.xml
<schema>

//Generic Schema Definitions
<documents name="base">
  <document name="XXX">
    ...
  </document>
</documents>

//derivations; you may have several of these sections (for
//servicecenter, assetcenter, user derivations, and so on)
<documents name="DERIVED_TARGET">
  <document name="XXX">
    ...
  </document>
</documents>
</schema>
```

Schema Attributes

The following describes all applicable document schema attributes.

<document>

This tag defines a document. The document may contain nested `<attribute>`, `<collection>`, and `<document>` tags.

A schema file should only define a single top-level document and its derivations.

The `<document>` tag can contain the following attributes:

Attribute	Description
name <i>(required)</i>	Uniquely identifies the document being defined. The name of the schema file must match the document's name. For example, the Problem document is defined in <code>Problem.xml</code> .
table	Defines the primary database table associated with this document. While not all document fields have to come from this table, the Primary Key (ID) for the document must reside in this table. This attribute is normally only defined by derived document schemas. That is, the derivations for ServiceCenter, AssetCenter, etc. must define where to get the document.

Nested <document> Tags

Top-level documents may include one or more nested documents. These children (or nested) documents may be defined in two ways.

The first way is to define nested documents in-place. For instance:

```
<document name="TopLevel">
  <document name="Child">
    <attribute name="x">
      ...
    </document>
  </document>
```

More typically, nested documents will reference a document defined in its own schema file. For instance:

```
<document name="Product">
  <document name="Vendor" />
</document>
```

Here the **Product** document contains a nested **Vendor** description. But because the nested **Vendor** document is defined to be empty, we assume that its definition should be looked up in the proper schema file (i.e. vendor.xml).

You can find nested documents by doing a search of the following type:

```
SELECT <Fields> FROM <NestedDocTable>
WHERE <joinfield>=<joinvalue>
```

The `joinfield` and `joinvalue` settings come from the schema's `<collection>` entry. For example:

```
<collection name="Assets">
  <document name="Asset" joinfield="lUserId" joinvalue="Id"/>
</collection>
```

The entry above defines a nested collection of assets that could appear within a parent "User" document. The `joinfield` and `joinvalue` specify that we want to find entries in the asset table whose "lUserId" field matches the parent table's ID field. (The parent's `joinvalue` is specified as a logical document field name).

If no "joinfield" or "joinvalue" are defined, the default is to use the parent table's ID field name as the join field.

<attribute>

The `<attribute>` tag defines a field within a document. Right now this tag can only appear within a `<document>` tag. All documents must define at least one mandatory attribute:

```
<attribute name="Id">
```

This attribute defines the unique key for locating document instances.

The `<attribute>` tag can have the following XML attributes:

Attribute	Description
name <i>(required)</i>	Uniquely identifies an attribute within a document.
type <i>(optional)</i>	Identifies the type of the field being defined. Possible values are: id, string, number, date, url This attribute is currently not used by the document manager. However, in the future it could be used to verify at run-time that a document is properly formed.
field	The name of the physical field to use in when building queries or updating the document table. This can be a simple name in the document's primary table, or it can be linked field name (AssetCenter only). <pre> <document name="Request" table="amRequest"> ... <attribute name="TotalCost" field="mTotalCost"/> <attribute name="Budget" field="Budget.Name"/> ... </document> </pre> <p>TotalCost is associated with the <i>mTotalCost</i> field in amRequest. Budget is associated with the linked field <i>Budget.Name</i>.</p>
link, linktable, linkfield, linktype, linkkey	These attributes work together to define how a field from a linked table should be accessed. Consider the following attribute in the Request document definition for AssetCenter: <pre> <attribute name="Budget" field="Budget.Name" link="lBudgId" linktable="amBudget" linkfield="Name"/> </pre> <p>Now consider a request to insert a Request document such as:</p> <pre> <Request> <Budget> 1999 IS Budget </Budget> ... </Request> </pre> <p>When the DocumentManager updates or inserts a Request document, the schema tells it to:</p> <ul style="list-style-type: none"> • search the linktable (amBudget) for an entry where the linkfield (Name) matches "1999 IS Budget". • use the link entry ID (lBudgId) to update the Request document table.

<collection>

The <collection> tag allows the nesting of collections inside a top level document. For example:

```
<document name="Request">
  ...
  <collection name="RequestLines">
    <document name="RequestLine"/>
  </collection>
</document>
```

A collection can only have one thing inside of it: a nested document.

This example shows a Request document with a nested collection of RequestLine documents.

Nested documents are found by doing a search of the following type:

```
SELECT <Fields> FROM <NestedDocTable> WHERE <joinfield>=<joinvalue>
```

The "joinfield" and "joinvalue" settings come from the schema's <collection> entry. For instance:

For instance, consider a list of assets owned by a user:

```
<collection name="Assets" joinfield="lUserId" joinvalue="Id">
```

The entry above defines a nested collection of assets that could appear within a parent "User" document. The joinfield and joinvalue specify that we want to find entries in the asset table whose "lUserId" field matches the parent table's Id field. (The parent's joinvalue is specified as a logical document field name).

If no "joinfield" or "joinvalue" are defined, the default is to use the parent table's Id field name as the join field.

ServiceCenter-Specific Attributes

Several attributes have been defined specifically for supporting ServiceCenter derived schemas. These are necessary for the following reasons:

- Documents should not be inserted directly into the ServiceCenter database. Instead, they should be created and updated by related EventServices calls.
- The basic elements of the schema DTD assumes a relational organization of data. ServiceCenter's non-relational database introduces some requirements.

Consider the following example or a derived Problem schema where ServiceCenter specific attributes are shown in bold:

```
<document name="Problem" table="probsummary" insert="pmo"
update="pmu">
```

```

<attribute name="Id" field="number"/>

    <attribute name="OpenTime" field="open.time"/>
    <attribute name="Status" field="status"/>
    <attribute name="AssignedTo" field="assignee.name"/>
    <attribute name="Priority" field="priority.code"/>

    <attribute name="Description" field="brief.description"
        insert="$ax.field.name" update="_null"/>
    <attribute name="Updates" field="update.action"/>
    <attribute name="Resolution" field="resolution"/>

</document>

```

The following attributes are used by SCDocManager, a derived DocManager class that is used by the SCAdapter:

Attribute	Description
insert	<p>This attribute ties a document to a specific input event. The attribute can be used in two ways.</p> <p>Within a <code><document></code> tag, the insert attribute names the event to use for inserting document instances.</p> <p>Within an <code><attribute></code> tag, the insert attribute names an event parameter name to use for a document field. If no insert attribute is defined, the default field setting is used instead.</p>
update	<p>This attribute ties a document to a specific update event. It can be used within <code><document></code> and <code><attribute></code> tags in the same way as insert.</p>

Note: A field, update, or insert setting with a value of `_null` tells the DocumentManager that the particular document element is not supported by the system.



Appendix C

JavaScript

This chapter contains JavaScript functions for a variety of processes.

Script Polling Manager

There are several classes in Archway that can be used as utilities to facilitate workflow between two or more systems. There are classes that provide basic message queuing (the Qman package) and there are classes that support polling. The following is an overview of the polling package.

Script Polling is a function that establishes a repetitive routine. A scheduler initiates a routine at a specified time as opposed to every n seconds. For example, a scheduler can run a routine at midnight, every Sunday night, at the end of the month or quarter or every hour (meaning the top of the hour, not 60 minutes from the last run).

You can define Script Pollers in each package. For example, the B2B and B2BServer packages have their own `scriptpollers.ini` files. At run time, Archway processes all `scriptpollers.ini` files in the registered packages.

File Initialization Format

The format of the initialization file is XML, located within the `...\getit\apps\common\` directory. The following is a sample for version 1.0, specifying two scripts to run. Note the `vpollShowTime` script.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!--=====
Name:    ScriptPollers.ini
Specifies a collection of scripts to run at periodic intervals
(seconds) and a parameter to pass to the script.
=====-->
<pollers>
  <poller>
    <name>vpollPmoSco2Q</name>
    <interval>4</interval>
    <parm></parm>
  </poller>
  <poller>
    <name>vpollShowTime</name>
    <interval>1</interval>
    <parm></parm>
```

```
</poller>
</pollers>
```

Writing A Polling Routine

To create a polling routine, write an FESI script and save it in the `...getit/apps/user/jscript` directory. Add the script name and polling interval to the `scriptpollers.ini` file in the archway bin directory. You can perform a quick test by adding the following line of code to the `ArchwayDebug` class:

```
ScriptPollingManager.test( m_archway );
```

Put the same functionality that is in the `test()` function into a more mainstream class (as discussed above).

Sample Polling Scripts

There are several polling scripts that exist in Get.It!, each beginning with the characters `vpoll`. They display the current time on the console (for illustrative purposes), read from ServiceCenter **eventin**, and are then sent to a **qman** queue in XML format (from eventin format). They are then received from a qman input queue (specified by arbitrary name) and send to a qman output queue (as would happen during transliteration). Receive from the qman output queue and insert back to ServiceCenter eventin.

The following is the simplest of polling scripts, as described above.

```
//simple poller script, displays current time of day
//two functions are defined
//start() - executes once
//run() - executes on the polling interval
Archway = Packages.com.peregrine.archway;
Message = Archway.Message;
Event = Archway.Event;

//start function .. can build parameters for run method
function start( msg )
{
    return msg;
}

//run function .. subsequent invocations
function run( msg )
{
    var date = new Date();
    writeln( ">>> The time is " + date.toLocaleString() );
    return msg;
}
```

Note: Sample JScript are available within the `...getit\apps\examples\jscript\` directory.

Read History Array

This function requests the `UniversalBrowserRead` privilege to enable it to read the array elements of the `History` object.

```
function openHistoryWindow(){
//open a new window
var w=window.open("", "historyWindow",
    "width=500,height=300,menubar,resizable");
var d=w.document;

//request a privilege
netscape.security.PrivilegeManager.enablePrivilege("
UniversalBrowserRead");

//output the browsing history as links in the new
for (var i=0;i<history.length;i++) {
d.write('<A TARGET="new" HREF="' + history[i] + '>');
d.write(history[i]);
d.writeIn('</A>');
}
d.close();

//return the new window
return.w;

//the privilege is automatically disable with function returns
}
```

Extract URL Argument

The following sample parses comma-separated `name=value` argument pairs from the URL query string. It stores the `name=value` pairs in properties of an object and returns that object.

```
//get arguments
function getArgs() {
var args=new Object();

//get query string
var query=location.search.substring(1);

//break at comma
var pairs=query.split(",");

for(var i=0; i<pairs.length; i++) {

//look for "name=value"
var pos=pairs[i].indexOf('=');

//if not found, skip
```

```

    if (pos== -1) continue;
//extract name
var argname=pairs[i].substring(0,pos);
//extract value
var value=pairs[i].substring(pos+1);
//store as property
args[argname]=unescape(value)

```

You can also use the `getArgs()` function to parse optional animation parameters from the URL.

```

//get arguments
var args=getArgs();
//if arguments are defined
if (args.x) x=parseInt(args.x);
//...override default values
if (args.y) y=parseInt(args.y);
if (args.w) w=parseInt(args.w);
if (args.h) h=parseInt(args.h);
if (args.dx) dx=parseInt (args.dx);
if (args.dy) dy=parseInt(args.dy);
if (args.interval) interval=ParseInt(args.interval);

```

Searchable Popup Window

Searchable Popup Windows (Lookup Window) are used for searching entries in a form. Implementing the Lookup field allows you to reuse it throughout the webapplication. To use this function you must add a `lookup` input field to any form. The following example illustrates an Employee Lookup.

```

<input label="User" field="UserId" displayfield="UserFullName"
type="lookup" readonly="true" balloon="Change User"
target-module="commonlookup" target-activity="employee"/>

```

The `label` is the label or name of the field. `field` is the hidden field that is to be returned to the server. The `displayfield` field is an optional display field name. The `type` field is the lookup type. The `readonly` field controls the edit/read function. The `balloon` field is for lookup help when the cursor is placed over an icon. The `target-module` field is for generic searching. The `target-activity` field is for searching activity.

Once the `lookup.getEmployees` function is in the code the `getX` and `selectX` scripts must be implemented. For generic activities this occurs in

`common\jscript\lookup.js`. The following is an example of this script for the

Employee search.

```
function getEmployees( msg )
{
return archway.sendDocQuery( "ac", "Employee",msg);
}
```

The `selectX()` script processes and returns both the selection field ID and the selection display field.

```
function selectEmployee( msg)
{
  var msgResponse = new Message();
  var strId = msg.get( "Id" );

  // Return selected ID as the searchable field
  var strField = user.get ( "_lookupfield" );
  msgResponse.set( strField, strId );

  // Return employee's Full Name as the display field
  var strDisplayField = user.get ( "_lookupdisplay" );
  var msgEmp = archway.sendDocQuery( "ac",
  "SELECT FirstName,LastName,FullName FROM Profile WHERE Id=" +
  strId, 0, -1 );
  msgResponse.add( strDisplayField, msgEmp.get( "FullName" ) );
}
```

This script returns the selected Employee ID as the Lookup Field and then returns the Employee Full Name field as the `LookupDisplay` field. These values update all fields in the calling browser window. The `lookup` field is a hidden field returned to the server when a form is submitted. The Lookup Display field is a text or entry field to display a user-friendly version of the selection.

Passing External JavaScript

If you want to pass a JavaScript function to the `archway.ini` file that is not in the `...\getit\apps\` directory then you must define the following path.

```
derivedscriptpath=<path...>
```

Inserting this line will cause Archway to seek alternative paths for user-defined scripts. This also becomes the path that Archway uses to locate “derived” scripts that override standard out-of-box scripts. If you define this in the `archway.ini` file then the new path replaces the `...\src\apps\user\jscript\` path. Original or out-of-box scripts reside in the same place.



Index

Symbols

\$(X) A-18
... 1-4
<action> A-14
 <back> A-15
 <home> A-15
 <link> A-15
 <submit> A-15
 TARGET A-15
<activity> A-4
<application> A-2
<attribute>
 insert B-6
 update B-6
<back> A-15
<columns> A-23
<component> A-34
<document>
 insert B-6
 update B-6
<entrytable> A-11
<field> A-25
 colspan A-13
 rowspan A-13
<fieldtable> A-11
<form>
 adding a field 4-12
<home> A-15
<html> A-35
<input> A-26
 checkbox A-29
 colspan A-13
 combo box A-28
 hidden field A-31
 radio button A-30
 rowspan A-13
 selection box A-28
 text area A-28
 text field A-27
<link image=X> A-19
<link TARGET> A-19
<link window="true"> A-19

<link> A-15
 attributes A-18
 hypertext link A-33
<listbox> A-10, A-24
<menu> A-10, A-18
<module> A-3
 access to A-3
 attributes A-3
<popupwindow> A-35
<submit> A-15
<table> A-20
 <column> A-20
 <link TARGET> A-20
 record A-20
 rows A-20
<target URL> A-4
 _null B-6

A

action property 2-11
activity
 changing the name 4-7
adapters
 AssetCenter 6-5
 e-mail 6-21
 JDBC 6-9
 LDAP 6-20
 ServiceCenter 6-2
 using log files to troubleshoot 6-25
 verifying connections 6-24
admin.xml
 updating for the JDBC adapter 6-14
apphead A-3
appmenu A-3

- archway architecture 4-1
 - building blocks 2-2
 - clients 2-3
 - diagram 2-2
 - document manager 2-10
 - executing queries against a system 2-10
 - how it works 2-3
 - internal architecture 2-5
 - query string 2-6
 - requests 2-6
 - weblications 2-11
 - XML 2-3
- archway.ini
 - updating for JDBC adapter 6-12
- archway.ini parameters 6-2, 6-5
- AssetCenter adapter 6-5
 - archway.ini parameters 6-5
 - setting feature links 6-7
 - troubleshooting the DB connection 6-7

C

- cascading style sheets 2-14
- changes
 - required steps 4-2
 - where to store 4-3
- child documents, *See nested documents*
- clients 2-3
- colspan A-13
- column
 - field to display A-21
 - headings A-21
 - productid A-21
 - select box A-22
- condition A-9
- CSS, *See cascading style sheets*

D

- debugging
 - script C-2
- displaylist A-22, A-29
- document manager 2-10, 4-17

E

- ECMA script 2-8
- e-mail adapter 6-21
- entry table A-11
- event handling in ServiceCenter 6-2

F

- feature links in AssetCenter, setting 6-7
- field table A-11

form

- changing contents 4-12
- create a menu of links A-18
- finding to edit 4-6
- image A-25
- input document A-7
- reusable components A-33
- static text A-25

form fields

- <entry table> A-10
- <fields> A-10
- <html> A-10
- <listbox> A-10
- <menu> A-10
- <plug in> A-10
- <table> A-10

form statistics

- using to find files 4-6

G

- Get.Answers!
 - modify hit list 5-3
 - modify site definition file 5-2
- getCatalog 2-14
- getOrderParameters 4-20
- getProduct 4-15

H

- hidden field A-31
- hit list
 - modifying 5-3
- HTML A-18, A-35
- hypertext A-35
- hypertext link A-33

I

- input document A-7

J

- JDBC adapter
 - adding 6-10
 - adding a new module 6-16
 - adding to the Settings page 6-14
 - calling a stored procedure 6-18
 - overview 6-9
 - updating the admin.xml file 6-14
 - updating the archway.ini file 6-12
 - validating the connection 6-13
 - verifying the system DSN 6-10
- joinfield B-3
- joinvalue B-3

L

- LDAP adapter
 - connecting 6-20
 - updating the archway.ini file 6-21
- log files 6-25

M

- module
 - adding 4-22
 - changing the name 4-7
 - removing from Get.It! 4-26

N

- nested documents B-2
 - finding B-3
 - in-place B-2
 - reference B-2
- null B-6

O

- onload property 2-14
- onload script 4-15

P

- param A-17
- popup window A-36
- presentation folders 4-4
- productid A-21
- Production Environment 4-44

Q

- query string 2-6

R

- regenerating web pages 4-2
- reusable form components A-33
- rowspan A-13

S

- schema 4-17
 - <attribute> B-3
 - <collection> B-5
 - <document> B-2
- attributes B-2
- document file B-1
- nested documents B-2
- ServiceCenter B-5
- structure B-1
- script
 - changing 4-19
 - debugging C-2
 - user-derived 4-19
- script pollers C-1
- scripting 2-8

scripts

- hard-coded queries 3-5
- SQL queries 3-5
- searchable popup window A-36
- sendDocInsert 3-5
- sendDocQuery 3-4
- sendDocUpdate 3-5
- ServiceCenter
 - derived schemas B-5
- ServiceCenter adapter 6-2
 - _event parameter 6-3
 - archway.ini parameters 6-2
 - event handling 6-2
 - troubleshooting the DB connection 6-4
- site definition file
 - modifying 5-2
- software
 - linking in to Get.It! 4-22
- stored procedures, calling 6-18
- subforms A-33
- submit A-15

T

- table 2-15
- tags A-1
- tailoring
 - basics 4-3
 - testing the user directory 4-8
- TARGET
 - param A-17
 - target-activity A-17
 - target-field A-17
 - target-form A-17
 - target-module A-17
 - target-url A-17
- TEXT A-17
- troubleshooting using log files 6-25

U

- user derived script 4-19
- user.xml 4-3
- user.xsl 4-22
- user-access A-3

V

- valuelist A-22, A-29

W

- wbuild 3-5, 4-2
 - parameters 4-4
 - XSL templates 4-22
- web pages
 - regenerating 4-2

- weblication 2-3, 2-11
 - additional functionality A-35
 - cascading style sheets
 - definition 2-14
 - ingredients 2-13
 - structure A-1
 - XSL layout templates 2-13
- windows
 - popup A-36

X

XML 2-3

XSL

- purpose of 4-22
- to learn more 2-13
- wbuild 4-22
- when to change 4-22

XSL layout templates 2-13