Get It!

Tailoring Guide
Release 1.1 SP2
April 2000

Contents

Weblication Reference

Document Schema DTD

Contacting Peregrine Systems

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.

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!	Chapter 4: Tailoring Get.It!
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
Contacting Peregrine Systems.	Appendix C: Contacting Peregrine Systems

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. If information is printed next to the window, it is important and you should pay special attention to it. For example:

Make sure your form statistics are displayed. See "Showing Form Information" on page 4-3 for instructions

Fig. 3-8 Adding a field to a form.

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 italics, 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, make sure you make note of the correct directory and replace the default path with the one that is correct for your system.

- File names are also shown in italics, 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 being discussed. The samples of code are not entire files; they are only 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 like **Get.Resources!** 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's 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's 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:
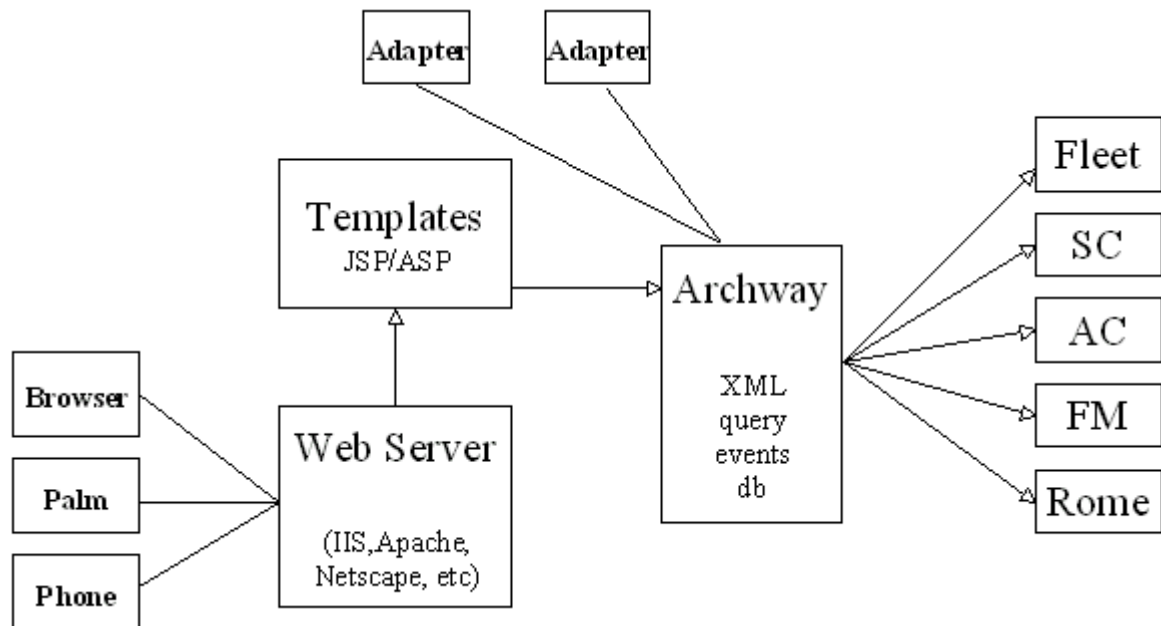
Fig. 2.1   The architecture

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 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 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 putting together logical documents.

The following diagram illustrates the internal Archway architecture.

```
                        ┌─────────────────────┐
                        │      Archway        │
                        └─────────────────────┘
                                   │
        ┌──────────────────────────┼──────────────────────────┐
        │                          │                          │
 ┌──────────────┐         ┌──────────────┐          ┌──────────────┐
 │ Script Runner│ ──────▶ │   Adapter    │ ◀──────  │  DocManager  │
 └──────────────┘         └──────────────┘          └──────────────┘
      │                         ▲                         │
   ┌──┴──┐                ┌─────┴─────┐              ┌─────┴─────┐
 (JS)  (JS)         ┌──────────┐ ┌──────────┐     (Schema) (Schema)
                    │ SCAdapter│ │ ACAdapter│
                    └──────────┘ └──────────┘
                         │            │
                    (sccl32.dll)  (aamapi.dll)
```
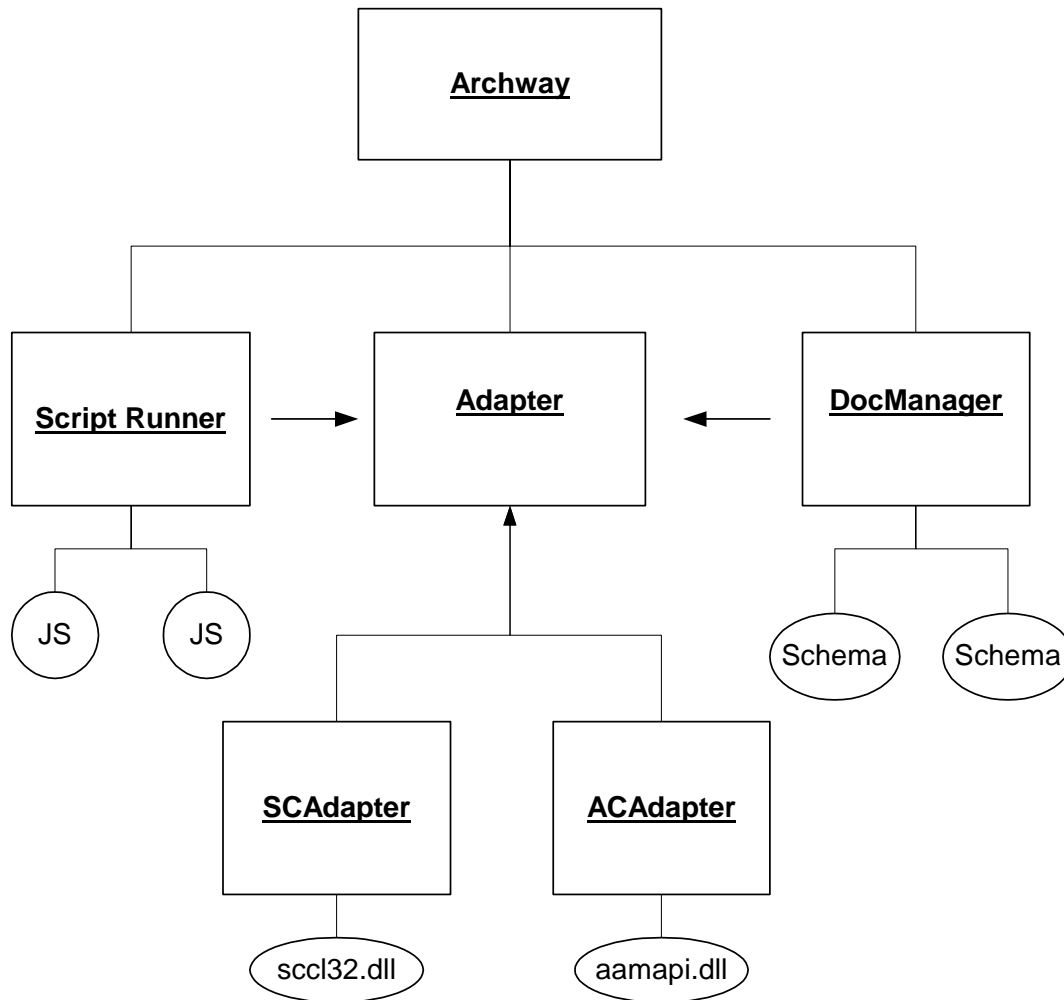
Fig. 2.2   Archway's internal architecture

As illustrated, 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, authentication, etc. Currently, adapters have been written for AssetCenter and ServiceCenter; however, we are preparing to write adapters for other Peregrine products and for external systems. The existing adapters utilize DLLs for each product to accomplish their communication.

Messages can be routed to a script interpreter hosted by Archway. The interpreter supports ECMAScript, which is a standardize based on several

originating technologies, the most well known being JavaScript (Netscape) and JScript (microsoft).

Finally, messages can be routed to the Document Manager component. This component reads special schema definitions that describe application documents for logical entities like a Purchase Request, a Problem Ticket, or a Product Catalog. The Document Manager uses the 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's 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, lets focus on two principal parts of a request:

Query String	This represents the *parameters* sent along with the URL for the HTTP connection.
	For instance, consider the following HTTP 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&param=value&param=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 introduction. See Chapter 4, "Tailoring Get.It!." for more details regarding the construction of Archway HTTP requests.

```
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** and **mPrice** 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=Hollo
```

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.
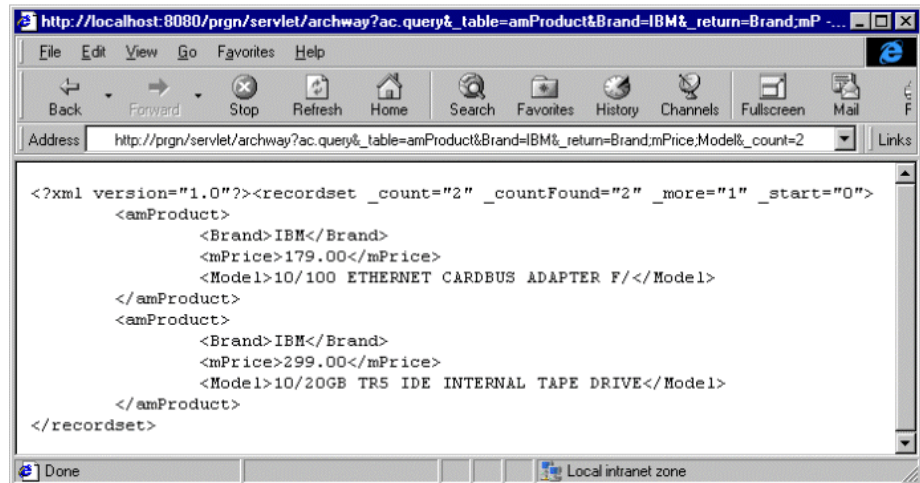
Fig. 2.3   Testing URLs from a web browser

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 reasons 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 a AssetCenter or 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 </brand>
  <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 ServiceCenter ticket. Defining this page in HTML might appear as follows:

```
<form action="http://prgn/archway?sc.pmo" method="GET">
   Name:   <input type="text" name="contact.name"> <br>
   Description: <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 though 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 = messanger.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, lets consider the Get.It! Weblications as an example. Below is a screenshot of a page in Get.Resources. The page shows a table with results from a catalog search.
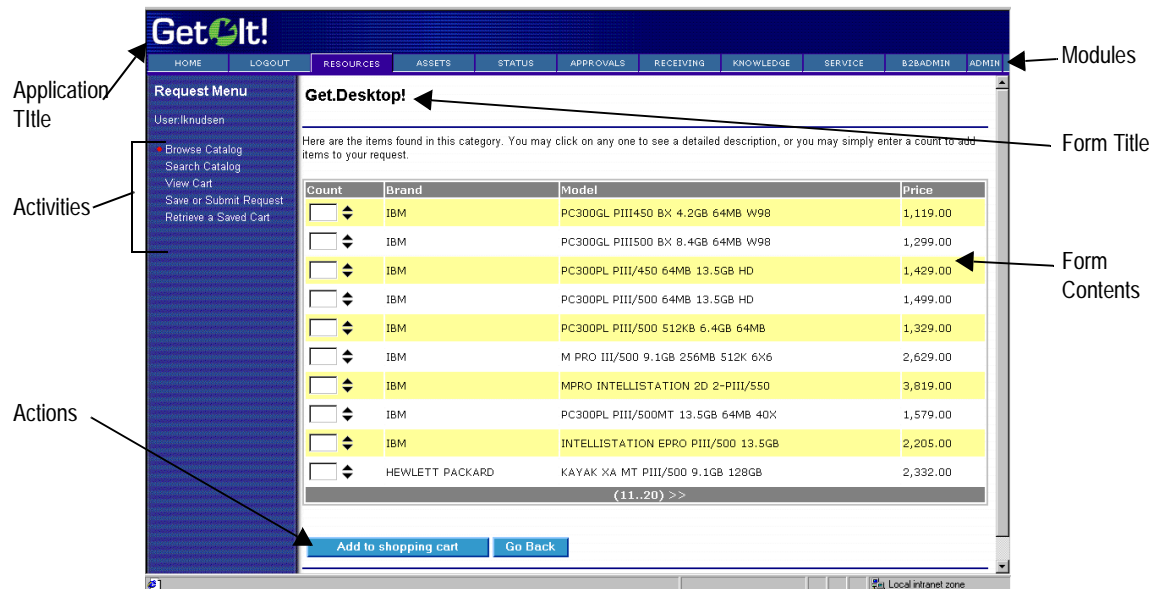
Fig. 2.4   Sample Get.It! Weblication window

.

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 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-xslt-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 the 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," etc.

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 in essence 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.

- 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>
```

Attached 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">
```

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

```
  <!-- 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"/>
```

This is an example of a nested document as a reference.

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

This is an example of a nested document in place.

```
    <!-- 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>
</documents>
```

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

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.

Using Schemas in a Weblication

In using Document and Schema support, "document" type archway messages are available to ECMA scripts. 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, we do not want hard coded queries in our 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 the queries that build up 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 simply 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 Peregine 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

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 the Appendix A, "Weblication Reference," at the back of this manual.
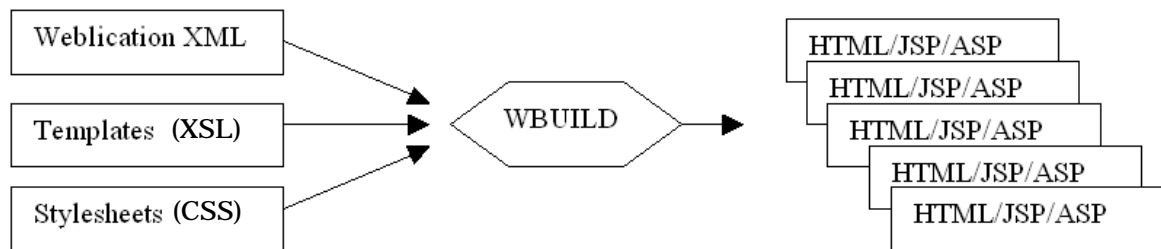
Weblication Toolset

Before doing any customization, it is useful 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	Define the layout and component construction rules for creating pages in a Weblication.
WBUILD	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* 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.

Showing 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 the Show form info" is set to true, a box is displayed on the left of the window as shown in the sample below:

Module= The name of the XML file in the *...getit/apps/* directory.
Activity and **Form**= Use these as search criteria to locate the exact form you want to change.
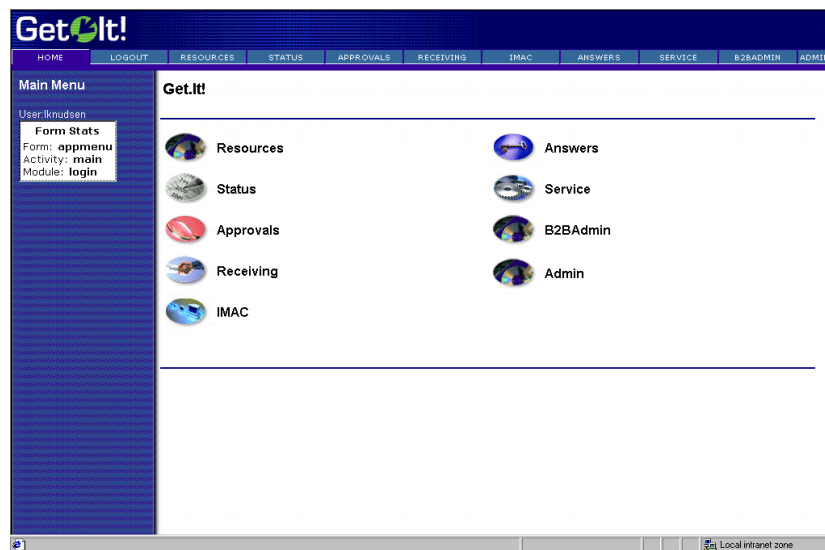

Fig. 4.2   Showing the forms information

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 for to change.

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

Debugging the Changes You Make

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 Get.It!'s Administration Module.

3. In the activities, click **Settings**.

4. In the section titled *General Execution Options*, set the *Debug logging* option to **true**.

5. In the section titled *ECMA Script Execution Settings*, set the *Debug Mode* option to **true**.

6. Click **Save** to save your changes (scroll down to below the settings table to find the Save button).

7. Be sure to set these options back to **false** before you release your changes your entire user base.

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 keep your changes safe, we have devised a method for you to use.

1. Create a directory called "user" within the directory where the original file resides. (So if you want to change the request.xml file in the *...getit/apps/* directory, you would create a directory called *...getit/apps/user.*) If you wanted to update a schema (directory *...getit/schema*), you would create a directory called *...getit/schema/user.* If you change a script, save the files in the directory you specified in the Get.It! Administration Module Settings.

2. Open the file you want to change. This could be a schema file, an application file, or a script.

3. Use the Save As command to save the file into the new "user" directory.

4. Make your changes and save the file.

5. If you have changed an application, tell Get.It! to use your new file instead of the one provided out-of-the-box by entering an import command in the *user.xml* file in the *...getit/apps/* directory.

```
<import href:"user/filename.xml"/>
```

where *filename* = the name you gave the file when you saved it into the "user" directory)

6. Run wbuild.

Information You Must Have

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

1. Display a command prompt. One method of doing this is to use Start>Programs>Command Prompt. Change the directory to C:>Program Files\GetIt\bin. (To change the directory, first ensure you are at 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** at the prompt and press **Enter**. The wbuild command will list all the processing it is going through. When you see "Done" the processing is complete. Minimize this window.

Changing the Contents of a Form

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 like:

Make sure you store you changes in a "user" directory. See "Where to Make the Modifications" on page 4-4 for details.

- **Title**

- **Instructions**

- **Fields**

- **Menus**

- **Tables**

- **Links**

- **Action buttons**

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

Adding a Field to a Form

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.

Make sure your form statistics are displayed. See "Showing Form Information" on page 4-3 for instructions

Fig. 4.3   Adding a field to a form

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*).

1. Open the *request.xml* file from the *...getit/apps* directory.

2. Use the Save As command to save the file into your *...getit/apps/user* 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>
```

4. 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"/>
    <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>
```

5. Save the modified file in the *…getit/apps/user/* directory.

6. Modify the user.xml file in the getit/apps/ directory to include an import of the modified file. See "Imports" on page A-2 for detailed information on entering an import tag.

7. After making this modification, run wbuild to regenerate the form. See "Running the wbuild Command" on page 4-5 if you need instructions.

   The modified screen in the browser is displayed below.

Fig. 4.4   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.

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.

When the Field is not Defined in the Schema

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:

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

Fig. 4.5   Adding information to a schema

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 word, 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/schema* directory.

2. Use the Save As command to save the file into your *…getit/schema/user* directory.

3. Find the "request" generic document definition, which is shown below:

```
<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 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/schema/user/request.xml* file.

7. After making this modification, run wbuild to regenerate the form. See
    "Running the wbuild Command" on page 4-5 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 different in that you can choose where to locate it via the Settings in the Administration module. The default is to look for user scripts in *…getit/jscript/user/*.

Changing a 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. 4.6   Changing jscripts.

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 4.6 above, the form is defined in the *submit* activity of the *request* module.

2. Open the *...getit/apps/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 jscript 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/jscript/* directory.

6. Save the file in the user directory you specified. The default is *...getit/jscript/user/*.

7. Within this file, find the following code:

```
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 resopnse
  msgResult.add( msgDepartments );

  return msgResult;
}
```

9. Save your changes.

10. This 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.

11. Within the function, 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.

12. Next, we query for the data of interest:

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

13. 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 );
```

14. 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 the Components
and Layout of a Weblication (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 better fit 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 Would 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.

In both cases 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 *templates/jsp/* directory. The directory also contains a file named *user.xsl*. This is where you can enter your own customization.

The *user.xsl* 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 into 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.

If you integrate a URL as a module, the product will be available to users in the menu bar and from the users main menu.

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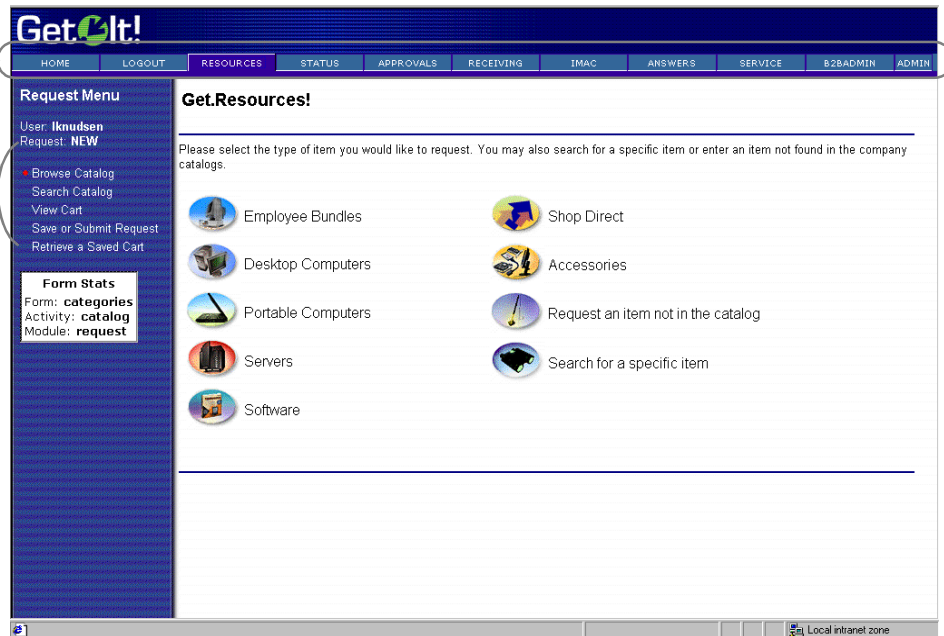

Fig. 4.7   Adding a URL as a module or as an activity

Adding the URL as a Module

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. Open the *e.xml* file (from the default path *C:\Program Files\GetIt\apps*).

3. Search the e.xml file for the following:

```
<!--==============================================================
    Sample link to external module. This generates a
    link to the Peregrine Company Web Site.
================================================================-->
  <module name="prgn">
    <title> Peregrine Home </title>
    <description
      image="images/smallglobe.gif"
      short="PRGN"
      long="Peregrine Home Page"
      target-url="http://www.peregrine.com"/>
  </module>
```

4. Copy these tags and paste them into the new file.

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 product. Update the "target-url" with the URL of the product you are integrating.

6. Save the new file in the *...getit/apps/user/* directory. Make sure it has an extension of ".xml".

7. Update the *user.xml* file to import the module into Get.It!. The user.xml file contains a number of sample imports which you can copy, such as:

```
<!--==============================================================
  Sample 1: Adding a new module
  This sample adds a module menu entry associated with another web site.
================================================================-->
 <Ximport href="samples/prgn.xml"/>
```

8. Enter the same information to have Get.It! import your new module. If you create a new XML file named *prgn.xml*, the import would be as follows:

```
 <!--==============================================================
   Peregrine Web Module
 ================================================================-->
 <import href="user/prgn.xml"/>
```

9. Save and close the *user.xml* file.

10. Run wbuild. See "Running the wbuild Command" on page 4-5 if you need detailed instructions.

11. Log out and 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 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/* 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. Update the *user.xml* file to import the module into Get.It!. The user.xml file contains a number of sample imports which you can copy, such as:

```
<!--==============================================================
   Sample 3: Adding a new activity
   This sample adds a new activity to the Service module to allow
   searching for an arbitrary ticket.
   =============================================================-->
<Ximport href="samples/service.xml"/>
```

9. Enter the same information to have Get.It! import your change. If you updated the service.xml file the import would be as follows:

```
<!--==============================================================
    Service with New Activity
   =============================================================-->
<import href="user/service.xml"/>
```

10. Save and close the *user.xml* file.

11. Run wbuild. See "Running the wbuild Command" on page 4-5 if you need detailed instructions.

12. Log out and back in to Get.It! and the new activity is available.

Adding a ServiceCenter or AssetCenter Feature as a New Module

Adding a new module requires you to copy an existing XML file, make your updates, and then import the new module through the *user.xml* file. 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 existing XML file (from 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 *...getit/apps/user/* 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/jscript/user/* 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 a jscript" on page 4-13 for instructions on updating a script file.

5. After you have created the new XML file, and the new jscript (if necessary), tell Get.It! to import the new module in the *user.xml* file. The *user.xml* file contains a number of imports, such as:

```
<!--================================================================
  Sample 4: Adding a new activity link
  This sample adds a new activity to the Resources module to display
  company policies regarding requests.
  ================================================================-->
<Ximport href="samples/request.xml"/>
```

6. You will need to enter the same information to have Get.It! import your new module.

   If you create a new module named *travel.xml* to handle travel requests, the module can be added as follows:

```
  <!--================================================================
    Travel Module
  ================================================================-->
  <import href="user/travel.xml"/>
```

7. Save the *user.xml* file. Saving it in the *user.xml* file allows you to load new versions of Get.It! without having all your changes overlaid.

   By importing the module, its web pages are automatically generated by wbuild. In addition, a link to your new module is automatically added to the Weblication header.

8. Run wbuild. See "Running the wbuild Command" on page 4-5 if you need detailed instructions.

9. Log out and back into Get.It! to see the changes you have made.

Modules can be removed from a Weblication by removing their imports in *user.xml* (if they are modules you created) or from the *e.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 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 4-10 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 the Contents of a Form" on page 4-6 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**. See "Running the wbuild Command" on page 4-5 if you need detailed instructions.

Appendix A
Weblication Reference

This chapter is a reference for the Weblication Extensible Markup Language Document Type Definition (XML DTD). The XML DTD is the high level XML language used to define the Get.It! Web Applications (*Weblications*).

Weblication Structure

The XML structure is made up of tags with supporting attribute and element information. All Weblications have the following basic structure:

```
<application>
  <modules>
    <module>
      <activity>
        <forms>
            <form>
            </form>
        </forms>
      </activity>
    </module>
  </modules>
</application>
```

That is, a Weblication is defined by an XML **application** entry.

The application is comprised of one or more **modules**, e.g., *service, request, approval, status,* and *receiving*.

Each module contains one or more **activities**. E.g., the request module has the following activities: *browse catalog, review shopping cart, submit order, retrieve saved cart.*

Each activity then 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.*

Imports

Weblication definitions may be divided up into separate files. Typically, each module, noted in the code by <module>, is saved in its own file. This makes it quick to identify which modules to include or exclude as necessary.

The **<import>** statement helps make this possible. This statement is defined as follows:

```
<import href="URL">
```

Where *URL* represents the file being imported. Currently, Get.It! supports URLs that represent an XML file stored in one of the following places:

- the same directory as the file doing the import
- a directory that is nested in the same directory as the file doing the import

For example, if you were adding an <import> tag to the user.xml file found in the ...getit/apps/ directory, you could import an XML file that is stored in either the ...getit/apps/ director or in an .../apps/user directory.

To see examples of the <import> tag being used, see the e.xml file, which contains imports similar to the following:

```
<import href="login.xml"/>
<import href="service.xml"/>
```

The user.xml file contains samples of importing a URL from a nested directory:

```
<import href="user/service.xml"/>
```

Weblication Tags

<application>

The <application> element is the starting point for defining a Weblication. It accepts the following attributes and nested elements:
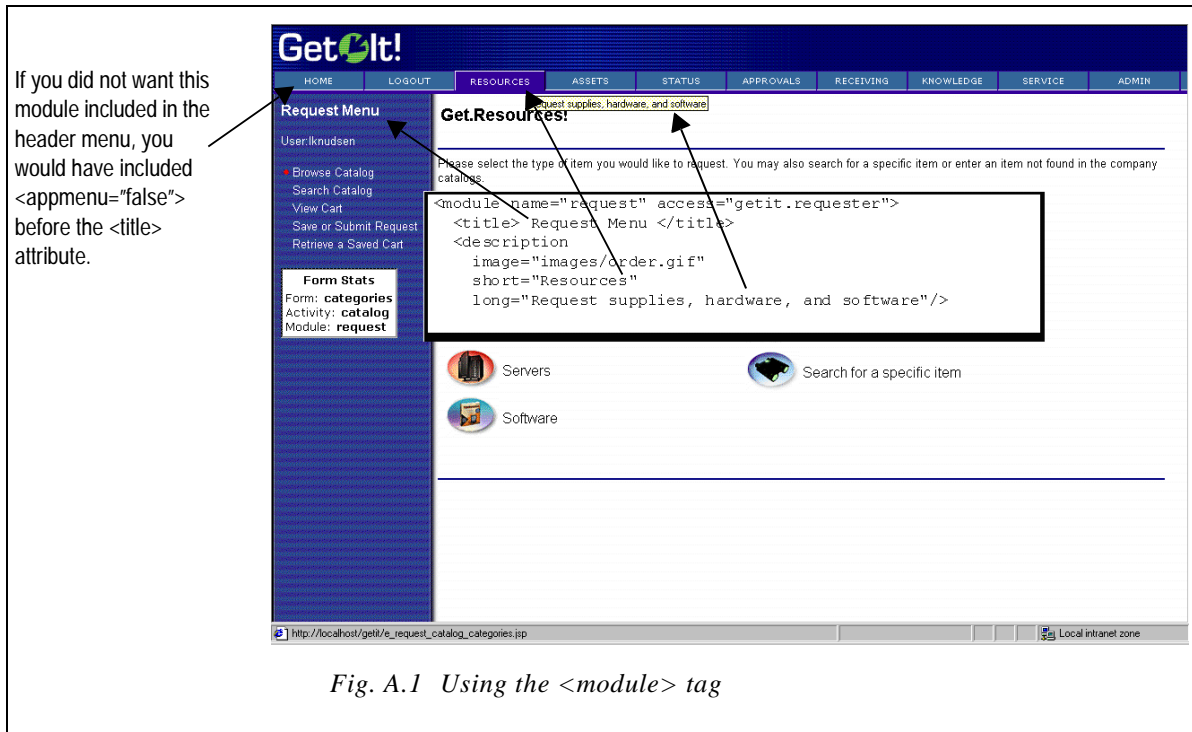
Attribute	Description
name *(required)*	A unique name for the Weblication. The name should be a single word starting with a letter.
<title>	Title used in the application's main menu.
<modules>	List of modules that make up the application.
frame	Specify whether you want the Get.It! banner to frame your windows. frame="true" causes the banners to display. frame="false" 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
name *(required)*	A unique name for the module. The name should be a single word starting with a letter.
access *(optional)*	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 that are not logged into the Weblication. all = The module can be accessed by all users who are logged into Get.It!
appmenu *(optional)*	Controls whether the module is included in the header shortcut menu. When set to *false* the module is not listed in the Weblication's header shortcut menu. The default is *true*.
apphead *(optional)*	Controls whether the module is included on the main menu. When set to *false* the module is not listed in the Weblication's main menu form. The default is *true*.
<title>	The title used to identify the module.

<description
    image="X"
    short="Y"
    long="Z">
	This element defines attributes that further describe the module.
	• 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 filetype.
	• 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 that is used as balloon help for links to the module.

<target URL> *(optional)*	Link a different module into this module.You can link any URL. See "Adding the URL as a Module" on page 4-18 of the Get.It! Tailoring Guide for details.
<activities>	List of activities that comprise the module.

See figure A.1 for a sample of how the <module> weblication tag and its attributes can be used.

If you did not want this module included in the header menu, you would have included <appmenu="false"> before the <title> attribute.

Fig. A.1   Using the <module> tag

<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 in order to make a request.

This element can contain the following attributes and nested elements:

Attribute	Description
name	A unique name for the module. The name should be a single word starting with a letter.
access	This optional attribute defines the name of a user *capability word* that is required in order to access the module. The default value is **anonymous**, meaning that the module may be accessed by any user, regardless of that user's profile capabilities. An **anonymous** module can be accessed by users that are not even logged into the Weblication.
<title>	Title used to identify the module.
<description image="X" short="Y" long="Z">	This element defines attributes that further describe the module. • The **image** attribute defines an image that can be used as a module 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 module. • The **long** attribute should contain a longer description that is used as balloon help for links to the module.
<target URL> *(optional)*	Link a module as an activity. You can link any URL. See "Adding a URL as an Activity" on page 4-19 of the Get.It! Tailoring Guide for details.
<activities>	List of activities that comprise the module.

See figure A.2 for a sample of how the <activity> weblication tag can be used.
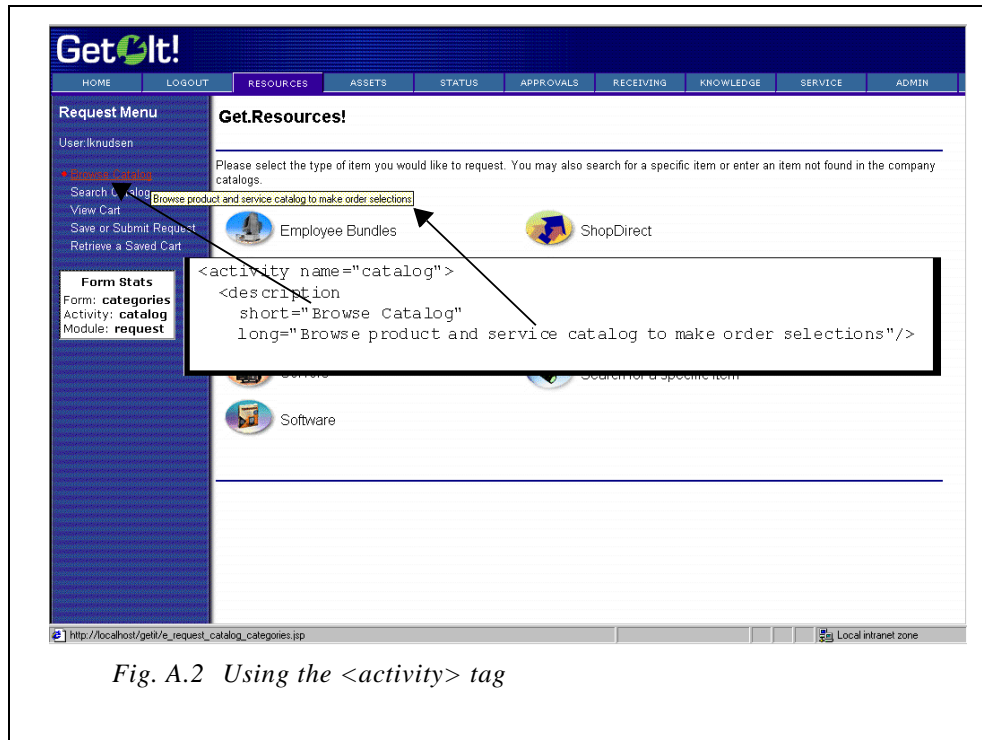
Fig. A.2   Using the <activity> tag

<form>

The <form> element is at the center of attention within a Weblication. This is where the specific contents for a screen are defined. For instance, the browser activity in the request module has a number of forms used to show product categories, product lists, product details, etc.

This element can contain the following attributes and nested elements:

Attribute	Description
name	A unique name for the module. The name should be a single word starting with a letter.
onload	Name of script to invoke before displaying the form. The Message returned by the script is used to populate fields in the form.
homepage	If set to *true*, the form is created to become the Weblication's homepage. Only one form should be given this attribute.
<redirect>	This element defines a condition that is evaluated before displaying the form. If the condition is *true*, an alternative form is displayed instead. See "<redirect>" on page A-9 for more information
<title> TEXT </title>	Title used to identify the form. See "TARGET" on page A-16 for more details on this attribute.
<instructions> TEXT </instructions>	Text giving the user instructions for the form.
<form fields>	One or more elements that make up the form, such as entry fields, labels, tables, menus, etc. See "form fields" on page A-11 for more information.
<actions>	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.
homepage	If set to *true*, the form is created to become the Weblication's homepage. Only one form should be given this attribute.

Important: 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.3 for a sample of how the <form> tag can be used.

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.

Count	Brand	Model	Price
	IBM	THINKPAD 390E PII/300 PE 4.3GB 64MB	2,399.00
	IBM	THINKPAD 390E PII/333 PE 6.4GB 64MB	2,699.00
	IBM	THINKPAD 570 PII/333MHZ 4.0GB 64MB	3,099.00
	IBM	THINKPAD 570 PII/300MHZ PE 4.0GB	2,699.00
	IBM	THINKPAD 570 PII/366MHZ 6.4GB 64MB	3,499.00
	IBM	THINKPAD 570 PII/366 6.4GB 64MB 56K	3,599.00
	IBM	THINKPAD 600E PII/300 AGP 4.0GB 24X	2,699.00
	IBM	THINKPAD 600 PII/300 5.1GB 64MB	3,699.00
	IBM	THINKPAD 600E PII/366 AGP 6.4GB 24X	3,399.00
	IBM	THINKPAD 600E PII/400 10GB 256K	3,799.00

(11..20) >>

The <redirect>, <table>, and <action> tags are explained in detail on the following pages.

```
<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.3 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 setup 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
TARGET	Defines the target location for the form. There are several ways to define TARGET location attributes. See "TARGET" on page A-16 for details on these attributes.
condition	This optional attribute defines 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.4   Using the <redirect> tag

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
<fields>	Groups one or more "field" elements, which include <input> and <field> elements. Sample fields include t**ext boxes, combos, check boxes, static text fields**, **input fields**, etc. 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.
<menu>	A menu of links.
<table>	A table whose rows are obtained dynamically at run-time from the form's input message.
<listbox>	A table whose rows are pre-defined within the Weblication.

Attribute	Description
<html>	Allows the insertion of any arbitrary HTML code.
<entry table>	A table that allows entries in one column and contains descriptions in another.
<plug in>	Allows you to plug in content from any web page that is accessible through a URL.

Fig. A.5 Using the table element of the form fields

<fieldtable>

A <fieldtable> element allows the creation of a nicely formatted table of entry fields. For example, the Request form is displayed using an <entrytable>. This element is used with 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.

Fig. A.6 The <fieldtable> tag in use.

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

Attribute	Description
colspan=N	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 colspan to specify that the field should take up both columns. For example: `<input type="textarea" field="descriptiom" colspan="2" ...>` The field above is given no label and is defined to span two columns. Therefore, the textarea takes up both the label and entry columns in a table. Typical values for colspan are 2 or 4.
rowspan=N	Allows a field to span more than one row in height. This is also typically used with textarea fields in a fieldtable.

<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
TARGET	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 TARGET for the actions is the *review* activity of the current module.
<submit>	Defines a submit button for a form. In the example above, the first submit entry displays a button with the caption *Add to shopping cart*. Clicking the button sends the user to the form's action target (the *review* activity). Any data entered in the form is sent along to the target form and will be available to the target form's *onload* script.
	Forms typically have one submit button. However, forms with more than one submit button can differentiate between them using the optional **name** attribute.

	For example, notice the second submit button. It also sends the user to the form's target destination (the *review* activity). However, the script of the target form can distinguish that is was invoked with the *Remove from cart* button because the button's name is sent along with the form. The script can check for this as follows:
	`if ( msg.get( "Remove" ) != "" )`
	`// form called with the "Remove" button ...`

Attribute	Description
<link>	Link actions are typically displayed by the Weblication just like any submit button. However, a link button offers a way to sent the user to any arbitrary TARGET destination. However, when a link is used, the form's data is not submitted to the target.
<back>	Creates a button that takes the user to the previous form.
<home>	Creates a button that takes the user to the home menu.

Fig. A.7   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 inter-dependencies. Therefore, the targets below

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 re-arranged.

The following are possible TARGET attributes:

Attribute	Description
target-form	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.
target-activity	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.
target-module	Links to the first form of the first activity in the named module.
target-url	Links to any URL. Anything that could be used in an HTTP `href` tag can appear here.
target-field	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: `<link target-field="VendorURL"> More information </link>` The target above is evaluated at run time by retrieving the **VendorURL** from the form's input document.
param	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: `<link target-form="catalog" param="Certification=Desktop"> Desktop Computers </link>` The link above passes a parameter named **Certification** with a value of *Desktop* to the target catalog form.

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 <br/>, , 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 <br>). Attributes inside HTML elements also must be quoted (e.g., <a href="x.htm"> rather than <a href=x.htm>).

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 following attributes and tags are supported in a menu:

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

Link Attributes

Attributes	Description
<link *TARGET*>	Defines the destination target of the link.
<link *image=X*>	Optional attribute that defines an image URL to use for the menu link.
<link *window="true"*>	If this optional attribute is set, the target of the link is displayed in a separate browser window.
<link *TEXT*</link>	The text used in the link. The <link> elements can 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

record	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:
	
	`<recordlist>`
	`    <Product>`
	`      <Brand> X </Brand>`
	`      <Price> 1 </Price>`
	`      <ProductId> 1356 </ProductId>`
	`      <nCount> 1 </nCount>`
	`    </Product>`
	`    ...`
	`  </recordlist>`
	
	To display a table with a list of products, the *record* attribute is set to *Product*. (This sample XML is used in the examples below)

rows	This optional attribute specifies 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.
`<link TARGET field=X>`	This optional table element is 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 *TARGET* attribute determines where the link is to take the user. The *field* attribute is used as a parameter passed to the target page. It is intended to uniquely identify the row. For example: `<link target-form="details" field="ProductId"/>` The link above comes from the *catalog* table. It creates row links that take the user to the *details* form. In addition, the **ProductId** 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.
`<column>`	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:

Column Types

Column Type	Description
Static text column	The default content of a column is static text. The *label* attribute specifies the column's heading. The *field* attribute defines the record field to display in the column. For example, `<column label=X field=X>`

Entry field column	These columns display a text *entry* field where the user can type in some text. The *label* and *field* attributes serve the same purpose as those of static text columns. The *key* attribute should contain the name of a record field that uniquely identifies each row in the column.
	
	`<column type="entry" label=X field=X key=X size=X>`
	
	The optional *size* attribute defines how wide to make the entry fields (in number of characters). For example:
	
	`<column label="Count" field="nCount" type="entry" key="ProductId" size="3"/>`
	
	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 **nCount** field from the table's record. The column uses each row's *ProductId* 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.

Column Type	Description
Select-box column (populated dynamically)	`<column type="select" label=X field=X key=X record=X valuelist=X displaylist=X >` 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: *valuelist* and *displaylist*. These are used to specify the name of the record field containing the choices for the select box. For example:

```
<column label="Project" type="select" field="ProductProject"
record="Project" valuelist="Id" displaylist="Title"/>
```

This column displays select boxes with a list of Project choices. For this to work, the form's input document should include Project entries such as:

```
<recordlist>
  <Project>
    <Id> 123 </Id>
    <Title> New Development 99 </Title>
  </Project>
  ...
</recordlist>
```

The selected choice is associated with the **ProductProject** field of the table's Product record. The choices displayed are determined by the **Title** field on the *Project* records, and the actual values submitted for each choice are those of the **Id** field in the *Project* records.

Column Type	Description
Select box column (populated statically	`<column type="select" label=X field=X key=X >` 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:

```
<column label="Approval" type="select" field="Approve">
  <option value="1"> Yes </choice>
  <option value="0"> No </choice>
</column>
```

This column displays **Approval** choices of *Yes* and *No*.

Column Type	Description
Image column	`<column label=X field=X>` This column displays an image. The image's URL is obtained from the specified field in the table's input record.
Radio button column	`<column label="Current employee" type="radio" field="Field1"/>` The *field* 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.

<columns>

It is possible to split a Weblication form into columns, as shown in the following example:

```
<columns>
  <column>
    Weblication elements for this column ....
  </column>
  <column>
    Weblication elements for this column
  </column>
<column>
```

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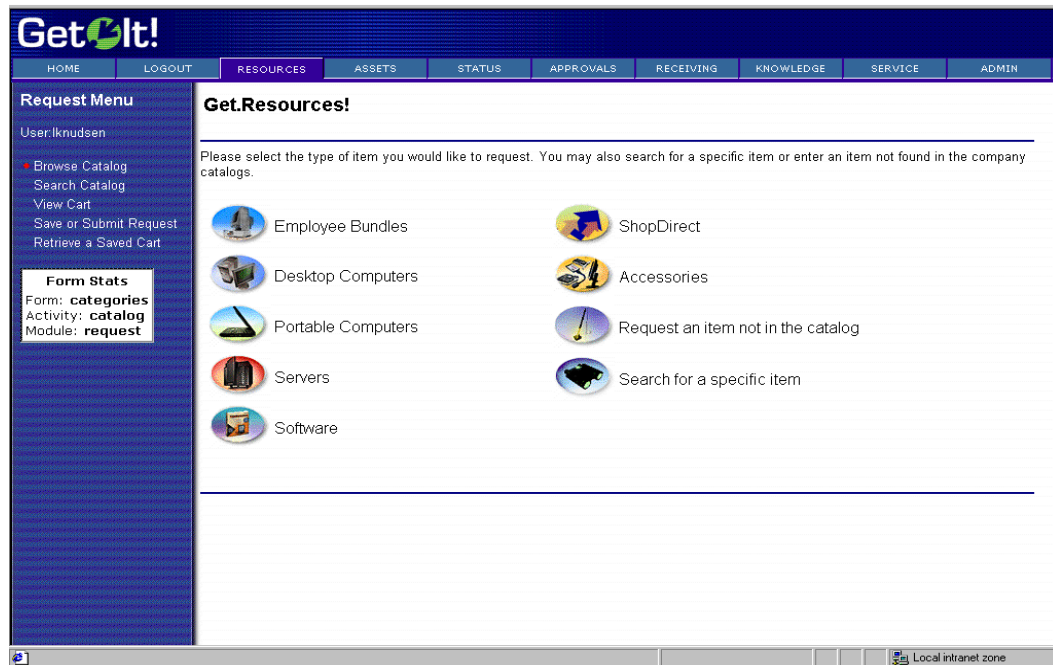
This code, when displayed in Get.It!, is shown below.

Fig. A.8 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 weblication. 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
\<heading\>	Defines the listbox headings. This element should be followed by one or more nested \<field\> elements that describe each heading.
\<row\>	Defines a row in a listbox. This should be followed by one or more nested \<field\> elements that are part of the row.
\<field image=X field=X\> Text \</field\>	A single element that may be placed in a heading or row cell. The optional image attribute may point to an image URL to display for the field. The field attribute may point to a field from the form's input document. Otherwise, the field displays its text contents.
\<input\>	You can enter any valid input element. See "\<input\>" 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 a <fields> parent element. The following attributes are supported:

Attribute	Description
label	This optional attribute specifies the label for the field.
field	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.
type	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.
<field> TEXT <field>	The value displayed in the field, displayed if no *field* attribute is already defined.

<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
label	This optional attribute specifies the label for the input field.
field	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.
type	Specifies the type of field.

Attribute	Description
value	This is an optional attribute. 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.
scope	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: *scope="user"* . 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 <input> elements for a field with user scope, the last value entered is always remembered.
required	If *true*, 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
label	This optional attribute specifies the label for the input field.
field	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.
type="text"	To create text entry fields, type should be set to "text". This is the default value.
value	This is an optional attribute. 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.
size	Optional attribute that defines the width of the text entry field in characters.

<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
label	This optional attribute specifies the label for the input field.
field	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.
type="textarea"	To create multiline text entry fields, type should be set to "textarea".
value	This is an optional attribute. 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.
rows	Number of rows in the textarea.
cols	Width of the textarea in number of characters.

<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
label	This optional attribute specifies the label for the select box.
field	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.
type="select"	To create text entry fields, type should be set to "text". This is the default value.
record	Specifies the record in the form's input document that contains the list of display and value lists.
valuelist	Specifies the field in the select box record that contains the values for each of the select choices.
displaylist	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 </choice>
  <option value="0"> No </choice>
</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
label	This optional attribute specifies the label for the checkbox.
field	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.
type="checkbox"	To create checkboxes text entry fields, type should be set to "checkbox".
value	Specifies the value that the checkbox field should have when the checkbox is selected.
<checkbox> **Text** </checkbox>	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
label *(optional)*	Specifies the label for the radio button.
field	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.

Attribute	Description
type="radio"	To create radio buttons, type should be set to "radio".
value	Specifies the value that the radio button should have when the radio is selected.
\<radio\> **Text** \</radio\>	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
field	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.
type="text"	To create text entry fields, type should be set to *text*. This is the default value.
value	This is an optional attribute. 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
\<link *TARGET*\>	Defines the destination target of the link. See "TARGET" on page A-16 for details about the TARGET attribute.
\<link *image=X*\>	Optional attribute that defines an image URL to use for the menu link.
\<link *window="true"*\>	If this optional attribute is set, the target of the link is displayed in a separate browser window.
\<link\> *TEXT* \</link\>	The text used in the link.
\<link field\>	

Reusable Form Components (Subforms)

It is common for a weblication 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, etc. 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>
```

The example above 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
onload *(optional)*	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 $$(X) expressions instead of using the form's input document.

Appendix B
Document Schema DTD

This chapter is a specification reference for defining schemas. 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"?>

<!--================================================================
  Name:   filename.xml
  Author: xxx
  Date:   xxx
================================================================-->

<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, etc.)
================================================================-->

<documents name="DERIVED_TARGET">

  <document name="XXX">
  ...
  </document>

</documents>


</schema>
```

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 *(required)*	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 Problem.xml.
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 *(required)*	Uniquely identifies an attribute within a document.
type *(optional)*	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).
	
	For instance:
	
	`<document name="Request"    table="amRequest">`
	`  ...`
	`  <attribute name="TotalCost" field="mTotalCost"/>`
	`  <attribute name="Budget"    field="Budget.Name"/>`
	`    ...`
	`</document>`
	
	  • **TotalCost** is associated with the *mTotalCost* field in amRequet.
	  • **Budget** is associated with the linked field *Budget.Name*.

Attribute	Description
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:

```
<attribute name="Budget" field="Budget.Name" link="lBudgId"
linktable="amBudget" linkfield="Name"/>
```

Now consider a request to insert a Request document such as:

```
<Request>
  <Budget> 1999 IS Budget </Budget>
  ...
</Request>
```

When the DocumentManager updates of inserts a Request document, the schema tells it to:

• 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	This attribute ties a document to a specific input event. The attribute can be used in two ways. Within a <document> tag, the insert attribute names the event to use for inserting document instances. Within an <attribute> 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.
update	This attribute ties a document to a specific update event. It can be used within <document> and <attribute> tags in the same way as insert.

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
Contacting Peregrine Systems

Contact one of the Peregrine Systems Customer Support offices listed here if you have questions about, or problems with, Get.It!.

For more information about Customer Support, check the support web site: *http://support.peregrine.com* Please contact Customer Support for an account on this site.

Note: Only the European Customer Support staff is multilingual and can provide technical support to customers in their native language.

North and South America

To get help immediately, call Peregrine Systems Customer Support at:

(1) (800) 960-9998 (North America only)

(1) (858) 794-7428 (North and South America)

Send materials that Peregrine Systems Customer Support requests to:

Peregrine Systems, Inc.
ATTN: Customer Support
12670 High Bluff Drive
San Diego, CA 92130

United Kingdom regional office

United Kingdom and South Africa
Peregrine Systems Ltd.
1st Floor
Ambassador House
Paradise Road
Richmond, Surrey,
United Kingdom TW9 1SQ

Phone: 0800 834770 (toll free)
or: (+44) (0) 181 334-5844

Fax: (+44) (0) 181 334-5890
E-mail: uksupport@peregrine.com

France regional office

France, Spain, Italy, Greece, and Africa (except South Africa)
Peregrine Systems
Tour Franklin-La Défense 8
92042 Paris La Défense Cedex, France
Phone: (+33) (0) (800) 505-100 (International Toll Free)
Fax: (+33) (0) (1) 47-73-11-61
E-mail: frsupport@peregrine.fr

Germany regional office

Germany and Eastern Europe

Peregrine Systems GmbH
Bürohaus Atricom
Lyoner Strasse 15,
60528 Frankfurt, Germany

Phone: (+49) (0) 69 67-73-4177
or: 0800ASSETCE (0800/2 77 38 23) (in Germany only)
Fax: (+49) (0) 69 67-73-4158
E-mail: psc@peregrine.de

Nordic regional office

Denmark, Norway, Sweden, Finland, and Iceland
Peregrine Systems A/S
Naverland 2, 12 SAL
DK-2600 Glostrup
Denmark

Tollfree in Denmark: (+45) 8030 7676
or: (+45) 4346-7676
Fax: (+45) 4346-7677
Customers in Norway, Sweden, Finland, Iceland
Phone: (+45) 7731-7776
E-mail: support.nordic@peregrine.com

Benelux regional office

Netherlands, Belgium, and Luxembourg

Peregrine Systems BV
Botnische Golf 9a
3446 CN Woerden
Netherlands
Phone: 0800 0230 889 (toll free in the Netherlands)
Phone: 0800 7474 7575 (toll free in Belgium and
Luxembourg)
E-mail: benelux.support@peregrine.com

Asia-Pacific regional offices

Australia   (800) 146-849
Hawaii   (1) (800) 960-9998
Hong Kong   (800) 90-8056
Japan   (0044) 221-22795
Singapore   (800) 1300-949 or -948
E-mail: apsupport@peregrine.com

Index

document file B-2
  nested documents B-3
  ServiceCenter B-7
  structure B-2
script
  changing 4-13
  user-derived 4-13
scripting 2-8
scripts
  hard-coded queries 3-5
  SQL queries 3-5
sendDocInsert 3-5
sendDocQuery 3-5
sendDocUpdate 3-5
ServiceCenter
  derived schemas B-7
software
  linking in to Get.It! 4-17
subforms A-33
submit A-15
support, contacting C-1

T
table 2-15
tailoring
  basics 4-3
TARGET
  param A-17
  target-activity A-17
  target-field A-17
  target-form A-17
  target-module A-17
  target-url A-17
technical support, contacting C-1
TEXT A-17

U
user derived script 4-13
user.xml 4-3
user.xsl 4-16
user-access A-4

V
valuelist A-22, A-29

W
wbuild 3-6, 4-2
  XSL templates 4-16
web pages
  regenerating 4-2
weblication 2-3, 2-11

cascading style sheets
  definition 2-14
  ingredients 2-13
  XSL layout templates 2-13

X
XML 2-3
XSL
  purpose of 4-16
  to learn more 2-14
  wbuild 4-16
  when to change 4-16
XSL layout templates 2-13