JACK Intelligent Agents®
WebBot Manual

Copyright

Copyright © 2001-2012, Agent Oriented Software Pty. Ltd.

All rights reserved.

US Government Restricted Rights

Trademarks

All the trademarks mentioned in this document are the property of their respective owners.

Publisher Information

Agent Oriented Software Pty. Ltd.

P.O. Box 639,

Carlton South, Victoria, 3053

AUSTRALIA

Phone:	+61 3 9349 5055
Fax:	+61 3 9349 5088
Web:	http://www.agent-software.com

If you find any errors in this document or would like to suggest improvements, please let us know.

If you find any errors in this document or would like to suggest improvements, please let us know.

The JACK™ documentation set includes the following manuals and practicals:

Document	Description
Agent Manual	Describes the JACK programming language and infrastructure. JACK can be used to develop applications involving BDI agents.
Teams Manual	Describes the JACK Teams programming language extensions. JACK Teams can be used to develop applications that involve coordinated activity among teams of agents.
Development Environment Manual	Describes how to use the JACK Development Environment (JDE). The JDE is a graphical development environment that can be used to develop JACK agent and team-based applications.
JACOB Manual	Describes how to use JACOB. JACOB is an object modelling language that can be used for inter-process transport and object initialisation.
WebBot Manual	Describes how to use the JACK WebBot to develop JACK enabled web applications.
Design Tool Manual	Describes how to use the Design Tool to design and build an application within the JACK Development Environment.
Graphical Plan Editor Manual	Describes how to use the Graphical Plan Editor to develop graphical plans within the JACK Development Environment.
JACK Sim Manual	Describes how to use the JACK Sim framework for building and running repeatable agent simulations.
Tracing and Logging Manual	Describes the tracing and logging tools available with JACK.
Agent Practicals	A set of practicals designed to introduce the basic concepts involved in JACK programming.
Teams Practicals	A set of practicals designed to introduce the basic concepts involved in Teams programming.

Table of Contents

1 Preface

The JACK Intelligent Agents® WebBot (WebBot) is a framework that enables the implementation of web applications based around JACK™ agents. Through the medium of WebBot, JACK agents can have a hand in the dynamic generation of web pages, thereby providing intelligent responses to user input.

By providing a detailed tutorial example and an explanation of the technical foundation of WebBot, this manual will help you to use WebBot effectively.

1.1 Intended audience

To use WebBot effectively, you will need a sound technical background. At the very least, you will need to be familiar with your computer's operating system, and know how to create/edit files, and compile/execute programs. You should also have a good grasp of HTML, the JACK Agent Language, and the Java programming language. The section *Prerequisite Background Information* provides some preliminary background information on the components that underpin WebBot.

The standard WebBot distribution comes with a tutorial example, including scripts for building and running the example. A good starting point would be to copy these files and adapt them for your own purposes. Of course, depending on your technical skills, you may prefer to read through this manual, and then build and deploy your application using your preferred program development / distribution method.

1.2 Overview of this manual

Chapter 1 introduces background concepts required to understand and use WebBot. The background material covers JACK, the Servlet API, Servlet Containers, JavaServer Pages and the WebBot architecture.

Chapter 2 describes how to develop and run a WebBot application. It begins with installation instructions and an overview of the typical directory structure of WebBot applications. This is followed by a description of how to set up Servlet Container configuration files. The next section describes the JACK definitions required to make your application interact with the WebBot layer. The chapter concludes by setting out the steps required to build and run a WebBot application.

Chapter 3 presents a simple tutorial example which takes you through the steps required to code, build and run a WebBot example. The tutorial example does not make use of *sessions*. Sessions are used to maintain separate user interaction threads.

Chapter 4 augments the Chapter 3 example so that it handles sessions.

1.3 Typographical conventions

To facilitate your reading of this manual, we have adopted certain typographical conventions:

- In the narrative sections of this manual, *italics* are used to denote sections of text that benefit from special emphasis (such as new terms, mandatory instructions, and critical concepts). Also, italics are used in the code examples to highlight comments.

- In the code examples, particularly significant code fragments are given in **bold text** for emphasis. These include WebBot-specific pieces of code, or fragments which are referred to in the main text.

- Samples of code and program output appear in `this typeface`.

1.4 Related reference material

In addition to this manual, you may need to consult:

1. the *JACK™ Intelligent Agents Agent Manual*;

2. the Tomcat 4.0.4 documentation, and

3. servlet/JSP-related material.

1.5 Further information

You may find useful information and updates at the Agent Oriented Software Pty.Ltd. web site, http://www.agent-software.com.

2 The JACK WebBot architecture

2.1 Purpose and functionality

WebBot is a framework which supports the mapping of HTTP requests to JACK event handlers, and the generation of responses in the form of HTML pages. Using WebBot, you can implement a web application which makes use of JACK agents to dynamically generate web pages in response to user input.

2.2 Prerequisite background information

The current version of WebBot extends the Java Servlet API v2.3 to enable interfacing with JACK. With WebBot, you can effectively embed a JACK application within a web server and have it respond to HTTP requests: for example, from a browser.

Before using WebBot, it is helpful to have an understanding of the major components it uses. Apart from the JACK Agent Language itself, the most important component is the Java Servlet API v2.3. For our purposes (i.e. HTTP requests), servlets are Java components which run within a web server and generate responses to HTTP requests. WebBot extends the Java Servlet API v2.3 to enable integration with JACK.

WebBot provides a mechanism for dynamically generating web pages based on the reasoning of JACK agents. Although it is entirely possible to write Java code which dynamically generates the HTML page to be returned to the client browser, there is a much more straightforward way of achieving the same end. WebBot makes use of JavaServer Pages (JSPs). JSPs enable the specification of the structure of the web page as a mix of HTML and *scriptlets*. Scriptlets are Java code fragments that specify how to generate the dynamic portions of the web page. A JSP compiler is used to compile the JSP page into Java code which will generate the web page at runtime.

JACK, the Servlet API and JavaServer Pages are outlined in the sections below.

2.2.1 JACK

JACK Intelligent Agents® is a Java-based *agent-oriented* programming environment. Agent-oriented programming facilitates the implementation of autonomous computational entities, or *agents*, which exhibit rational decision-making behaviour. The autonomous nature of these agents makes them ideally suited to solving problems of a distributed and real-time nature. Web-based applications by their very nature are distributed, and should ideally provide real-time response.

Through the medium of a *web server*, web applications have been used successfully to service client requests over the web. However, the programming languages used to implement such web applications (e.g. PHP or ASP) provide little or no support for the implementation of

intelligent reasoning behaviour. The JACK Agent Language, on the other hand, has been designed from the ground up to support the encoding of intelligent reasoning processes into its agents. This, coupled with the autonomous nature of JACK agents, means that it is well-suited to the task of providing an intelligent back-end to web-based applications.

Before designing and implementing a web-based application using the JACK Agent Language, you will need to have a fairly thorough understanding of the JACK Agent Language and its capabilities. This can be acquired by attending the training courses offered by Agent Oriented Software Pty. Ltd., and by reading the *Agent Manual*. Note that an understanding of JACK Agent Language *threads* is integral to making full use of the flexibility offered by WebBot.

2.2.2 Servlet API

A servlet is a Java component which runs within a servlet container , such as Tomcat (used in this manual). Servlets conform to the HTTP request/response communication model, in which the client submits a request and then receives a response to that request. WebBot servlets extend the HTTP-specific aspect of the Servlet API. Servlets are Java objects and have a `service` method which has a `request` and a `response` parameter. The `request` parameter contains the data sent by the client. HTTP-specific servlets can support HTTP methods such as `GET` and `POST`, and hence can access information such as that provided in HTML *forms*.

When the user types in a URI (Uniform Resource Identifier) which is handled by a particular servlet, the Web server which hosts that URI will invoke the servlet in question. For example, to invoke the `calculator` servlet hosted at `www.webbot.example.com`, the user would enter the following URI:

`www.webbot.example.com/calculator`

A web page containing an HTML *form* which is to be processed by the `calculator` servlet, could contain the following *form element*:

```
<form method=get action="/calculator/appform">
 ...
</form>
```

where `appform` is used by the `calculator` servlet to determine how to process the form. This will be illustrated in the chapter titled *Tutorial Examples*.

2.2.3 JavaServer Pages (JSP)

JavaServer Pages (JSPs) use XML-like tags and Java scriptlets to specify how to generate the content of a web page. JSPs are compiled into Java; the Java code is used to generate the web page at runtime. Although it is possible to generate the complete response page within a servlet without using a JSP file, JSPs provide a means of separating the page's logic from its presentation to the user. This simplifies the design and implementation of dynamically-

generated web pages, because the static portions can be hard-coded as HTML in the JSP file and the dynamic portions included as Java scriptlets.

When a request object is received from the client, the servlet sets various parameters that are passed back in the response object. These parameters can be referenced from within the JSP file (within scriptlets). Each scriptlet (a piece of Java code between "`<%`" and "`%>`") is executed, and if it is an expression beginning with an "=" sign followed by a parameter name (e.g. `=parameterName`), the parameter value is inserted in place in the JSP file. These parameter values are interleaved with the static portions of HTML to form a page which is passed back to the client.

Typically, before the application is deployed, each JSP file is converted into a `.java` file which is then compiled into a `.class` file for inclusion in the servlet.

The example below illustrates a very simple JSP file which displays an email address obtained from the servlet. The scriptlets are **bolded** and explained in the comments (demarcated by "`<!--`" and "`-->`").

```
<!-- Static HTML code -->
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
      charset=iso-8859-1">
</head>
<body bgcolor=#FFFFFF marginwidth=0 marginheight=0 topmargin=0
leftmargin=0>

<!-- The variable, address, is assigned the value of the parameter,
     emailAddress, from the response from the servlet. -->
<% String address = (String)request.getSession().getValue("emailAddress");
%>

<!-- The value of the variable, address, is inserted in place. -->
<p>The email address is: <%=address%></p>

<!-- Static HTML code -->
</body>
</html>
```

In the above example, if the email address were `info@gadget-software.com`, the output would be:

```
The email address is: info@gadget-software.com
```

2.2.4 Servlet Container

WebBot should work with any Servlet Container implementing the Java Servlet API v2.3 . Tomcat 4.0.4 is one such implementation and the one with which WebBot has been most extensively tested.

Tomcat is the Servlet Container that is used as the reference implementation for Java Servlets and JavaServer Pages (the latter is called "JASPER" in Tomcat 4.0.4). It has been developed as part of the Jakarta Project (http://jakarta.apache.org/).

2.3 Overview of the JACK WebBot Architecture

This section outlines the general architecture of the WebBot framework in terms of the major components that constitute a deployed WebBot application. The processing steps and information flow that characterise a running WebBot application will then be described, followed by an overview of the major classes which make up WebBot.

2.3.1 The architecture of a JACK WebBot application

The figure below shows a high level view of the architecture of a general WebBot application. It is a three tiered architecture, with the Servlet Container at its base. The Servlet Container interacts with the client application, accepting HTTP requests and passing them on to the appropriate servlet in the WebBot layer. The WebBot layer performs some data marshalling, then passes on the request as a JACK *event* to the appropriate agent in the JACK Application layer. The agent's response is passed back through the three layers and is routed to the client as a new web page. The role of each layer is described in more detail below.
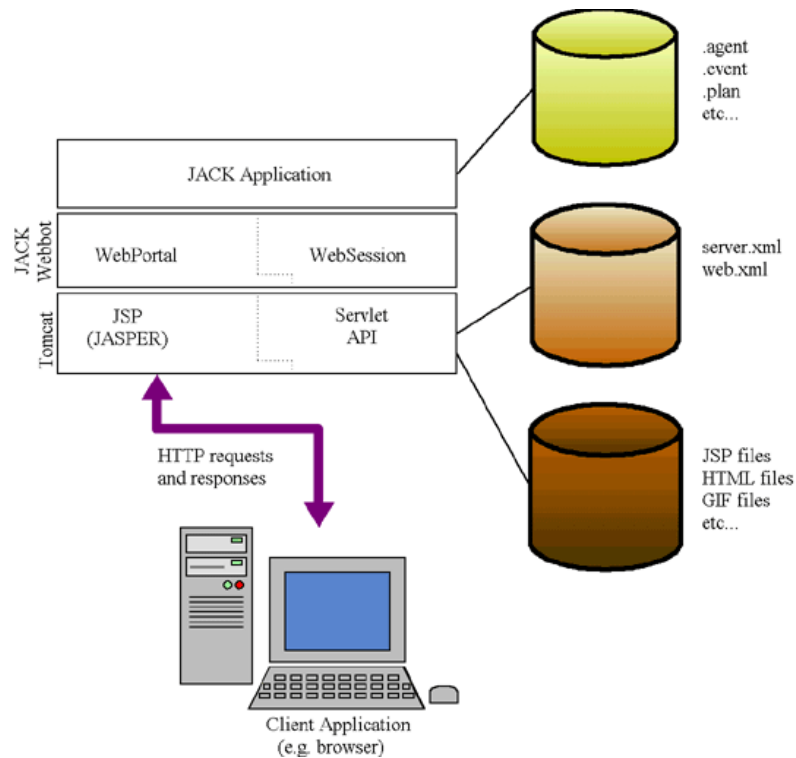
Figure 2-1: JACK WebBot Architecture

2.3.1.1 Layer 1 – Servlet Container

The Servlet Container implements the Servlet API and a JSP compiler. When the Servlet Container receives an HTTP request, it determines which servlet, if any, should handle it and

invokes the servlet in question, passing it the HTTP request. If the URL pattern of the request does not map to any servlet, the Servlet Container will respond to the client with a "404" error, indicating that the requested URL does not exist (this response page can be tuned for a particular application). Assuming that a servlet was found for the URL in question, the Servlet Container will generate an HTML page to return to the client; this page will be created by instantiating the JSP file returned by the servlet. Typically the JSP file will contain scriptlet-based references to various parameters assigned by the JACK Application layer.

2.3.1.2 Layer 2 – JACK WebBot

The WebBot layer extends `HttpServlet`, providing a framework for accessing the Servlet API from the JACK Agent Language. To this end, it provides the following functionality:

- If one does not exist already, the WebBot layer creates an agent of the appropriate type to handle the incoming request (the agent is specified by the `type` parameter in the configuration file `web.xml`).

- Converts the incoming HTTP request into a `WebRequest` event.

- Provides supporting classes to enable the JACK Application to post the event (using `postEventAndWait`).

- Provides support for managing *sessions*. Sessions allow the application to maintain *separate* interaction threads with one or more clients. For example, imagine a situation in which the user is using two windows to browse a beliefset managed by your WebBot application. Both beliefset browsing pages have a "Next" button which displays the next beliefset element matching the user's search key. Clearly, when the user clicks on "Next" in either window, your WebBot application needs to ensure that it responds with the beliefset element which pertains to the correct browser window, i.e. the *session* in question. Sessions are also used to manage requests from separate clients on the internet.

- Provides methods for accessing and setting parameters in the HTTP request object.

- Invokes the servlet class (typically compiled from a JSP definition) to issue the HTTP response. This response page can also be a page that indicates that the agent failed while handling the request (the filename is specified by the `noservice` parameter in `web.xml`), or that the application does not have an agent to handle the request (specified by the `nohandler` parameter in `web.xml`). Servlet Containers can be configured to dynamically compile JSP pages into java classes. These java classes are then loaded and run to produce the final output page. However, as WebBot does not have its own internal JSP compiler (unlike most Servlet Containers), JSP pages used by WebBot must be precompiled.

This layer is described further in the section *JACK WebBot Servlet*.

2.3.1.3 Layer 3 – the JACK application

This is the layer in which you will place all of your JACK code. This layer defines how your application will respond to incoming events. Generically speaking, the application will set a

number of response parameters and it will return a JSP file to be instantiated by the Servlet Container.

2.3.2 The dynamic behaviour of JACK WebBot applications

In the previous section, the functionality of each architectural layer was outlined, and the runtime behaviour of each layer was explained. This section will look at the processing steps followed by WebBot applications, and relate those steps to the three layers and the interactions between those layers.

In a typical interaction between a client and a WebBot application, the Servlet Container receives an HTTP request from the client, selects a servlet based on the URL pattern of the HTTP request and passes the request to the appropriate WebBot servlet. WebBot converts the request into a `WebRequest` event. This event is posted and will trigger one of the JACK agent's session-selection plans. This sets off a computational sequence culminating in the assignment of values to variables referenced in the JSP response file, i.e. the JSP file to be used by the Servlet Container to generate a response page for the client. A reference to that JSP file is then returned to the Servlet Container, which executes the scriptlets in the JSP file and sends the new page to the client.

These steps are described in more detail below. Because the JACK Application layer extends the WebBot layer and invokes methods within that layer, the combined behaviour is described in the one section.

2.3.2.1 Layer 1 – Servlet Container.

The interaction begins when the Servlet Container receives a HTTP request from the client. The complexity of this request depends on your application; for example, it might have been generated by an HTML *form* and could contain a sequence of attribute / value pairs. The configuration file `web.xml` defines the mapping between a given URL pattern and the servlet which handles HTTP requests matching that URL pattern. The Servlet Container uses this mapping to pass the request to the appropriate WebBot servlet.

2.3.2.2 Layers 2 and 3 – WebBot and the JACK application.

The WebBot class, `WebPortal` , implements the `javax.servlet.Servlet` interface by extending the class `javax.servlet.http.HttpServlet`. When an incoming HTTP request is received from the server, the `WebPortal` checks to see whether there is an agent to handle it. If there is not an agent, then one is created. This request-handling agent (termed the "root" agent) has special status: it is responsible for dispatching an appropriate `WebSessionRequest` event, based on the make-up of the `WebRequest` event.

All WebBot applications are structured as shown in the figure below. The `WebPortal` servlet takes the incoming HTTP request and uses it to generate a `WebRequest` event. This event should trigger the session-selection plan in the root agent. The session-selection plan determines which "session" agent should handle the request. It then uses that agent's `createSessionRequest` method to create a `WebSessionRequest` event. This event will then trigger one of the session agent's plans which, all being well, will set various parameters in the

response object and will determine which JSP file the Servlet Container should use to generate the response page.
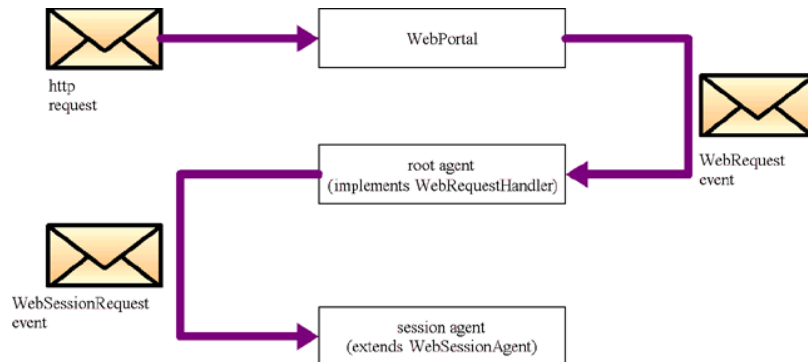
Figure 2-2: WebBot Execution Structure

2.3.3 JACK WebBot servlet

2.3.3.1 Major JACK WebBot classes

public class WebPortal extends HttpServlet

The `WebPortal` class is a `HttpServlet` that handles HTTP requests. The servlet is configured with the parameters `type` and `name` in the file `web.xml`. These parameters specify the class and instance name of the agent that handles incoming requests. When the `lookupHandler` method is invoked, if there is no agent of the specified `name`, the `WebPortal` creates one of the specified `type` with the specified `name` (specified in `web.xml`). When the `doGet` or `doPost` method is invoked, the `WebPortal` creates a `WebRequest` event and posts it to the agent using `postEventAndWait`.

public interface WebRequestHandler

The application's request-handling agent must implement this interface. The interface has one method, `public boolean handle`.

public class WebSessionAgent extends Agent implements HttpSession

The `WebSessionAgent` class implements `HttpSession` and provides methods for handling `WebSessionRequest`s, and other session-related activities such as adding a "`jsessionid`" to a URL (`sessionURL(String url)`) and storing values to be used in the returned JSP page (`putValue(String key, Object obj)`).

2.3.4 Sessions

Sessions form part of the Servlet API. Typically, they are used to manage separate client interactions with the server. WebBot provides support for the implementation of sessions through the provision of the `WebSessionAgent` and `WebSessionRequest` classes. The Servlet API provides support for tracking sessions through the use of "cookies". However, this method of tracking sessions is fairly limited. For example, if the client session occurs on a machine which is shared by others (e.g. in an internet cafe environment), then it is feasible for a new user to pick up on the previous user's session. Another drawback of cookies is that they are restricted to one session per user (i.e. browser).

A safer method of tracking sessions is to use URL rewriting. A session identifier is stored in a hidden attribute on the web page. This is then written into the URL before sending the HTTP request back to the server. This is the preferred method when using WebBot. If you store the session id in the attribute "`jsessionid`", and include that in the URL, then the `WebPortal setup` method will extract it and store it in the parameter "`id`". `WebPortal` will also pick up the "`referrerid`" and store it in the parameter "`refid`". This can be used to determine if the web page has a referrer, and so can be used in most cases to prevent someone from dropping in on a session (unless, of course the person has written their own browser).

3 Developing and running JACK WebBot applications

This chapter describes how to develop and run a WebBot application, beginning with the installation instructions for WebBot. Subsequent sections in this chapter cover the steps required to develop an application. The final section describes how to run the application.

To develop and deploy a WebBot application, you will need to define the Java Servlet API v2.3 specified in the `web.xml` configuration file. This is covered in the section *Configuring The Servlet Container*. The Servlet Container needs to be supplied with the set of JSP , HTML and ancillary files required by your application, described in the section *JSP and Ancillary Servlet Container Files*. Your JACK application will make use of some WebBot classes and will follow certain conventions which are explained in the section *JACK Definitions*. Finally, you will learn how to build and run your WebBot application in the section *Building and Running a JACK WebBot Application*.

3.1 Installation instructions

If you have problems installing WebBot, please AOS technical support.

The installation process for WebBot is simple and straightforward:

1. Install JACK in the directory of your choice.

2. Include all of your Servlet Container's `.jar` files in your `CLASSPATH`

Note that the examples in this manual can be found in `aos/jack/examples/webbot`

3.2 Typical Application Directory Structure

Figure 3-1: Standard WAR layout

The figure above shows the directory structure for a typical Web Application Archive (WAR). The Web Application Archive file is used to package up the web application so that it can be deployed on multiple platforms. The WAR format is shown in the above figure. The root directory contains the files that are served up to the client browser (e.g. HTML, JSP, CSS, JS and GIF files). If required, these files can be partitioned into a subdirectory structure. The `WEB-INF` directory contains the application's Web Application Deployment Descriptor, the file `web.xml`. `WEB-INF` also contains the `classes` and `lib` directories. The `classes` directory contains the application's Java class files. The `lib` directory contains JAR files that house library files used by your application.

Your WebBot application should be organised as shown under the root directory in the figure below. The `web` sub-directory corresponds to the root of the WAR structure shown in the figure above. The scripts are held in the root directory and the source files are located in the `src` sub-directory.

Figure 3-2: Typical JACK WebBot Application Directory Structure

3.3 Configuring the Servlet Container.

Configuring the Servlet Container to serve up a particular WebBot servlet requires that the file `web.xml` be configured. This section only covers the main parameters required for WebBot. You should probably use the `web.xml` file that came with your installation and edit the parameters specified below. The information provided below will suffice for most cases, however, for more detailed information, please consult your Servlet Container's documentation.

3.3.1 Setting up `web.xml`

Typically, most of the parameters in `web.xml`, can be left unchanged. Hence, the most straightforward approach to developing a new WebBot application is to take a copy of `web.xml` from your installation, and edit the parameters described in this section. Should your particular application require that you alter parameters not covered in this section, then refer to the documentation of your particular Servlet Container implementation.

The `web.xml` configuration file defines the *servlets* that constitute your application. It contains two major defining elements:

- The `<servlet>` definition is by far the largest, and contains the servlet's initialisation parameters.

- The `servlet-mapping` defines the mapping between the URL patterns and servlets, i.e. which servlet to run, given a particular URL.

The structure and function of the `<servlet>` and `servlet-mapping` elements are outlined below. Note that all of the elements prefixed by `<param-name>` are WebBot-specific. They occur in a defining element of the following form (N.B. bolded values would be replaced with the actual parameter name and value that pertains to your application):

```
<init-param>
  <param-name>xxx</param-name>
  <param-value>yyy</param-value>
</init-param>
```

3.3.1.1 The `servlet` element

Each servlet definition is delimited by "`<servlet>`" and "`</servlet>`". The key parameters of the servlet definition are as follows:

- `<servlet-name>` - the unique identifier for this servlet.

- `<servlet-class>` - the class that implements the WebBot servlet (e.g. `aos.web.webbot.portal.WebPortal`). The class is located in `jack.jar`.

- `<param-name>type` - the class name of the request-handling JACK agent. WebBot will create a "root" agent of this type to handle the incoming requests.

- `<param-name>name` - the instance name of the request handling agent. WebBot will bind this to the instance of the "root" agent that it creates.

- `<param-name>nohandler` - the class name of the class to invoke when the `WebPortal` cannot create an agent to handle the request. Typically, this class is defined as a side effect of the compilation of a JSP file created for this purpose.

- `<param-name>noservice` - the class name of the class to invoke when the agent handling the request fails (in the JACK Agent Language sense of the term "fail") to complete its processing of the event. Typically, this class is defined as a side effect of compilation of a JSP file created for this purpose.

- `servlet-mapping` - each servlet mapping definition is delimited by `<servlet-mapping>` and `</servlet-mapping>`. The `<servlet-mapping>` maps each `servlet-name` to the `url-pattern` that invokes it. This `url-pattern` is used to trigger the appropriate JACK plan for dealing with the request. In the example below, the `froboz` servlet handles URLs of the form `/froboz/*`.

```
<servlet-mapping>
  <servlet-name>froboz</servlet-name>
  <url-pattern>/froboz/*</url-pattern>
</servlet-mapping>
```

- `port` - this is the IP port used for JACK communication. You will only need to include this if your application has multiple agent processes communicating over the JACK communications network (dci). This parameter is passed in as the command-line argument "`-dci.new`".

- ns - the name and port of the JACK name server to use. As with port, this is a dci-specific parameter; it is passed in as if by the command-line argument "-dci:ns".

```
<init-param>
  <param-name>ns</param-name>
  <param-value>nowhere:9000</param-value>
</init-param>
```

3.4 JSP and ancillary Servlet Container files

 In order for a JACK agent to return a response to the client's browser, there must be some way of generating an HTML page for it. You are free to use any means at your disposal, for example, you could write Java code that dynamically generates HTML pages to be returned to the client browser. WebBot does not mandate how this should be done, but the most straightforward way is to use JavaServer Pages to generate the response pages. These allow you to specify the structure of the web page as a mix of HTML and scriptlets. Scriptlets are Java code fragments that specify how to generate the dynamic portions of the web page.

The particular mix of JSP, HTML and other files (e.g. .gif files) depends very much on your application requirements. Nevertheless, you will usually need to reference data items computed by the JACK agent that handled the client request. To do this, your JSP file contains scriptlets that access the values of the relevant JACK variables. The example below illustrates this. It accesses the value of the variable cost and assigns it to the String totalCost. The variable totalCost can then be referenced throughout the JSP file within scriptlets.

```
<% String totalCost = (String)request.getSession().getValue("cost"); %>
```

3.5 JACK definitions

As outlined in the section *The Dynamic Behaviour of Jack WEBBOT Applications*, the WebPortal servlet converts an incoming HTTP request into a WebRequest event; this event will trigger the root agent's session-selection plan, which will then select the "session" agent that should handle the event. The session agent's createSessionRequest method then creates a WebSessionRequest event that triggers one of the session agent's plans, culminating in the setting of various variables in the response object.

3.5.1 Servlet and JACK WebBot classes to import

There are a number of prerequisite classes that your WebBot must import:

1. the HTTP-specific request and response functionality of the Servlet API,

2. the JACK Agent Language, and

3. the WebPortal, WebRequest, WebRequestHandler, WebSessionAgent and WebSessionRequest classes.

To import the classes listed above, include the set of `import` statements shown below.

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import aos.web.webbot.portal.WebPortal;
import aos.web.webbot.portal.WebRequest;
import aos.web.webbot.portal.WebRequestHandler;
import aos.web.webbot.session.WebSessionAgent;
import aos.web.webbot.session.WebSessionRequest;
```

3.5.2 Agent, plan and event definitions

This section outlines the JACK definitions that you will need to include in order for your application to interact with the WebBot layer. At the very least you will need to define:

1.   an agent that `implements WebRequestHandler`, and

2.   an agent that `extends WebSessionAgent`.

The former requirement ensures that you have a *root* agent that can route incoming `WebRequests` to the appropriate *session* agent; the latter requirement ensures that your application can instantiate *session* agents. The two requirements can be satisfied by the one agent definition, i.e. by defining an agent type that `extends WebSessionAgent` and `implements WebRequestHandler`. Note that you would only adopt this approach if your application did not make use of sessions; this being the case, it would only need a single *root* agent and a single *session* agent, and so they may as well be one and the same agent. The first tutorial example adopts this approach (see *The Simple Calculator Example*).

The following two sections describe the definitional components and features that the *root* and *session* agents need to incorporate.

3.5.2.1 The root agent

The *root* agent is responsible for routing the incoming request to the appropriate session agent. This is done by invoking the appropriate session agent's `createSessionRequest` method. The `WebSessionRequest` so created will then trigger one of the session agent's plans.

The root agent should have the following components:

•   An `event` definition that `extends WebRequest`. When posted, this event must invoke the `WebRequest setup` method; this method sets up the `WebRequest` so that it holds the important fields passed through in the `HttpServletRequest`. A fairly generic example of such an event definition is shown below. The `WebSessionAgent a` and `Event e` are used to store the session agent and the `WebSessionRequest`.

```
public event WebDispatch extends WebRequest {

    public WebSessionAgent    a;
    public Event              e;

    #posted as
    select(WebPortal p, HttpServletRequest q,
           HttpServletResponse r) {
      setup(p, q, r);
    }
}
```

- A `handle` method that does a `postEventAndWait` of the `WebRequest` event that you defined (e.g. `WebDispatch` in the above example). The example below is fairly generic. It relates to the `WebDispatch` event defined above; the only non-generic parts are those shown in bold (they relate to the identifiers chosen for the `event` definition above).

```
public boolean handle(WebPortal p,
                      HttpServletRequest q,
                      HttpServletResponse r) {
   WebDispatch wd = dispatch.select(p, q, r);
   if (!postEventAndWait(wd))
      return false;
   if (wd.a == null || wd.e == null)
      return false;
   return wd.a.postEventAndWait(wd.e);
}
```

- At least one session-selection plan. This plan `handles` your application's specialisation of the `WebRequest` event (e.g. `WebDispatch` in the example above). It should store the session-handling agent instance in the `event`'s `WebSessionAgent` data member (e.g. `wd.a` in the above example). It should also invoke `createSessionRequest` and store the returned value in the `event`'s `Event` data member (e.g. `wd.e` in the above example). If the root agent's name is `DispatcherAgent`, then the following plan would minimally satisfy the requirements of session selection. It assumes that the root and session agents are one and the same; thus, `findAgent` merely returns the root agent (effectively returning `(WebSessionAgent) getAgent()`)

```
plan SelectSession extends Plan {

    #handles event WebDispatch ev;

    body() {
      ev.a = findAgent();
      ev.e = ev.a.createSessionRequest(ev);
    }

    #uses interface DispatcherAgent a;

    WebSessionAgent findAgent() {
      return a;
    }
}
```

3.5.2.2 Session agent

The *session* agent is responsible for dealing with the client request, setting any required values in the response object and returning a reference to the appropriate response page to be sent by the server to the client. A session agent should `extend WebSessionAgent` and should invoke `WebSessionAgent`'s constructor. Minimally, it requires a plan that `handles WebSessionRequest` events and sets the `response` data member. This is illustrated in the example below.

```
plan SimpleJSPResponse extends Plan {

  #handles event WebSessionRequest ev;
  #uses interface WebSessionAgent me;

  body() {

    try {
      me.response(ev, "froboz/froboz.jsp");
    }
    catch (Exception e) {
      System.err.println("No good path: " + ev.orig.path);
      false;
    }
  }
}
```

Note that this material is covered in the tutorial examples presented in the last two chapters.

3.6 Building and running a JACK WebBot application

To build and run a WebBot application, you will need to define a few environment variables, compile your application's JSP files and the JACK files, and finally run the Servlet Container. The environment variables are much the same for both building and running an application; any exceptions to this rule will be highlighted. The sequence of steps required to build and run the application is listed below.

1. Setup your environment as required by your particular Servlet Container implementation.

2. Precompile the JSP files. Consult your Servlet Container's documentation on how to achieve this with your particular installation.

3. Next, from within the *parent* directory of the JACK source files, prepare the `$CLASSPATH` to enable compilation of the JACK agents. When compiling the JACK agents,`$CLASSPATH` should additionally include the path to your JACK installation (e.g. `jack.jar`).

4. Compile the JACK agents as follows:

    — `java aos.main.JackBuild -r $*`

5. Start your Servlet Container.

These steps are encapsulated in scripts in the tutorial example in the section *The Simple Calculator Example*. You may find it useful to use these as a starting point for developing your own WebBot build and run scripts.

4 Tutorial example: simple calculator

4.1 Introduction

This chapter will take you step-by-step through the creation and running of a simple WebBot application. The example is fairly trivial in its use of the JACK Agent Language; the goal of the example is to demonstrate how to use WebBot, *not* how to program in the JACK Agent Language. The example can be found in `aos/jack/examples/webbot/calculator`

The simple calculator will then be extended in the section *Extending the Calculator to Handle Multiple Sessions*. An effort has been made to limit the need to consult other documents, but depending on your computing environment, you may need to consult the documentation that comes with the various third party components used by WebBot (e.g. the installation instructions for your Servlet Container).

This WebBot example implements a very simple four-function calculator. All of the calculator functions are handled by a single JACK plan, `FormResponse`. Note that if this were a real application it would be preferable to handle each calculator function using a separate plan.

To use the calculator, enter two numbers into the text fields and then press a button to select the operation to be applied. The JACK agent computes the result (which can be an error message if for example a division by zero occurs) and displays it to the user via the medium of a new JavaServer Page.
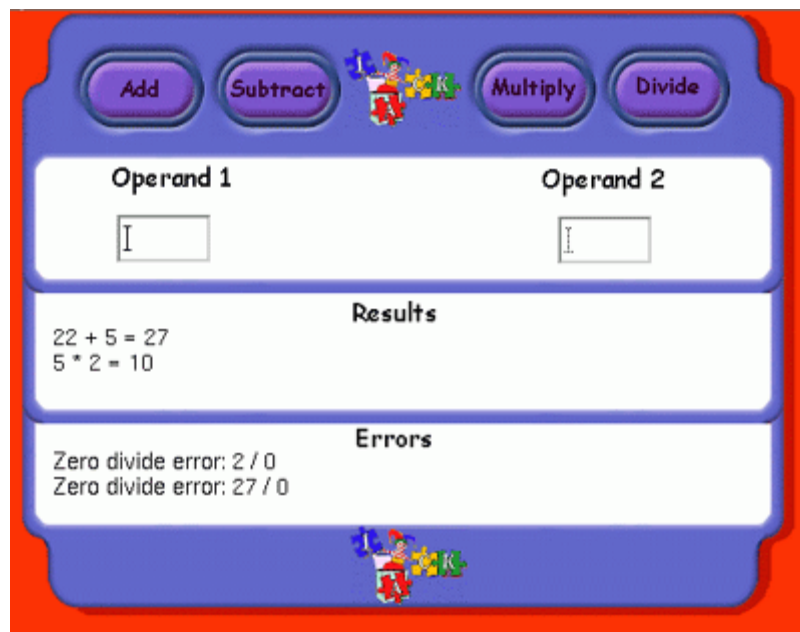
Figure 4-1: Screen Snapshot of the Calculator Web Page

The calculator device consists of a top row of four buttons, with three panes below. The top pane consists of two text input fields, each accepting a sequence of arbitrary text characters, up to a maximum length of 5. The left field is used to enter the first *integer* operand; the right field takes the second operand (also an integer). Once the two operands have been typed in, the user should then click on one of the four buttons at the top of the calculator. The WebBot application will then apply the arithmetic operation, denoted by the button, to the two operands. If the operands are valid arguments to the chosen arithmetic operator, then the result is displayed in the middle pane (which also contains the result of the previous computation, if any). Otherwise, if the operands are invalid, then an error message is generated and displayed in the bottom pane (which also contains the previous error message, if any).

If the operator is "Divide", the agent also checks to see whether the second operand is zero. If it is, an error string is returned and displayed in the bottom pane. The agent does not check the types of the operands. If you enter non-integers into the operand fields and select an operator, the application will simply bring up a page telling you that there is no such service (rather than specifically flagging it as a type violation).

In the remaining sections, you will learn how to:

1. Set up your environment so that you can compile and run the WebBot application, in *Setting up the Build and Run Scripts*.

2. Configure the Servlet Container for your application, in *Configuring the Servlet Container*.

3. Create a JSP file that will embody the logical structure of the web page, including the parameters returned by the JACK agent, in *Creating a JSP File*.

4. Define a JACK agent that makes use of the infrastructure provided by WebBot, in *Defining DispatcherAgent Agent*.

4.2 Installing the tutorial example

4.2.1 WAR! What is it good for?

We have decided to package the WebBot examples as Web Application Archives (WAR files). The Java Servlet Specification version 2.3 defines a web application as: '... a collection of servlets, html pages, classes, and other resources that make up a complete application on a web server. The web application can be bundled and run on multiple containers from multiple vendors.' This statement refers to one of the major advantages of packaging a WebBot application as WAR files. WARs are a delivery mechanism which is portable across different Servlet Containers, running on diverse operating systems.

The WebBot example code has build scripts that build the source code and package up the resultant class files and static web content into a WAR file. The example WebBot applications have the basic directory structure shown in the next figure.
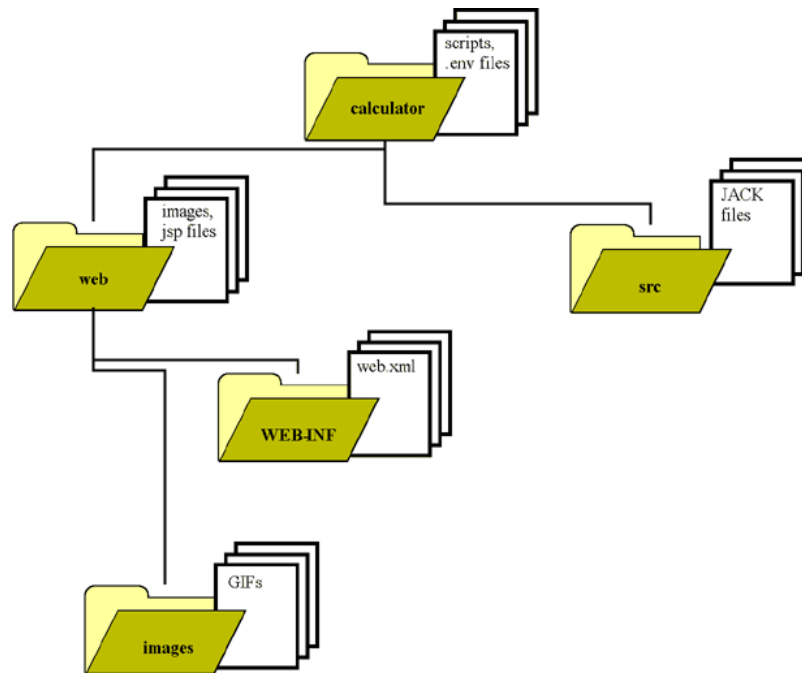
Figure 4-2: Directory Structure for the Calculator Example

4.3 Setting up the build and run scripts

This section describes how to modify the build and run scripts so that they reflect the structure of your computing environment. The calculator example is built by executing the script, mkwebapp. The mkwebapp script will also compile all JACK files into Java and compile all .java files into .class files. Note that servlet containers often dynamically compile JSP files into class files on demand. However, WebBot must have all JSP files it uses precompiled into classes before the web application is launched. The mkjsp script achieves this and is invoked from the mkwebapp script.

4.3.1 The build scripts, `mkwebapp` and `mkjsp`

The build script, mkwebapp, sets up a number of environment variables, invokes mkjsp to compile the JSP files, and then invokes the JACK builder, JackBuild. The mkwebapp script sets environment variables from the servlet.env and webapp.env files. You should not need to edit the mkwebapp file itself. The *.env files will however usually require modifications to match your own setup.

The servlet.env file defines three variables required by external scripts. They are JSP_COMPILER, SERVLET_CLASSPATH and SERVLET_CONTAINER_WEBAPPS. These refer respectively to the CLASSPATH definitions required for compiling JSP pages, the Servlet code and the directory in which web applications reside in your particular Servlet

Container setup. The `servlet.env` file delivered with the WebBot examples contains details suitable for those using Tomcat 4.x as their Servlet Container. If you are using Tomcat 4.x then the only variable which needs changing should be the TOMCAT_HOME variable at the top of the file.

Environment variables related directly to the web application itself are stored in `webapp.env`. These include variables for the name of the web application, the directory in which JACK is installed and the path to the directory in which you are developing your application.

The build script and default *.env files (suitable for Unix systems) are reproduced below:

4.3.1.1 The web application environment file, `webapp.env`

```
# Name of WebBot web application
WEBAPP_NAME=calculator

# Directory where you have installed JACK
JACK_HOME=$HOME/Agent_Software/

# Directory where you are developing your WebBot web application
WEBAPP_DIR=$JACK_HOME/aos/jack/examples/calculator
```

4.3.1.2 The Servlet Container environment file, `servlet.env`

```
# If the line below does not refer to the location of your Tomcat
installation,
# alter it so that it represents the path to the Tomcat directory.
export TOMCAT_HOME=/misc/jakarta-tomcat-4.0.4

LIB=$TOMCAT_HOME/lib

# Jar file containing classes for implementation of servlet API
SERVLET_JAR=$TOMCAT_HOME/common/lib/servlet.jar

SERVER=$LIB/webserver.jar

# Servlet XML parser
XML=$TOMCAT_HOME/common/lib/xerces.jar

# The next three variables are the only variables used externally
# from this script.

# CLASSPATH required to compiled jsp pages to java.
JSP_COMPILER=$LIB/jasper-compiler.jar:$LIB/jasper-runtime.jar

# CLASSPATH required to compile classes using the servlet API
SERVLET_CLASSPATH=$SERVLET_JAR:$SERVER:$XML

# Location of webapps directory in your servlet container.
# your web application (packed as a web archive (WAR) file) is copied
# into this location.
SERVLET_CONTAINER_WEBAPPS=$TOMCAT_HOME/webapps
```

4.3.1.3 Script file, `mkwebapp`

```
#!/bin/sh
source ./servlet.env
source ./webapp.env

# Alter the next line so that it references your JACK installation.
cp $JACK_HOME/lib/jack.jar lib

# The next line invokes the script which compiles the JSP files.
./mkjsp

# The line below defines the Java CLASSPATH variable.
export CLASSPATH=$WEBAPP_DIR/src:$SERVLET_CLASSPATH:$JACK_HOME/lib/
jack.jar:$JSP_COMPILER:$CLASSPATH

# Clean out any previous Jack compilation.
java aos.main.JackBuild -c $WEBAPP_DIR/src
java aos.main.JackBuild -c $WEBAPP_DIR/web
# The following lines invoke the Java compiler and build the JACK
# agents.
echo building in $WEBAPP_DIR
java aos.main.JackBuild -r $WEBAPP_DIR/src
java aos.main.JackBuild -r $WEBAPP_DIR/web
```

The `mkjsp` script also sets environment variables from `servlet.env` and `webapp.env`. However the line in the script which invokes the JSP compiling will need to be modified if you are not using the Apache project's Jasper JSP compiler.

4.3.1.4 Script file, `mkjsp`

```
#!/bin/zsh

source ./servlet.env
source ./webapp.env


WEB_DIR=$WEBAPP_DIR/web

# $JAVA_HOME needs to point to the location of your JDK. Alter the
# line below accordingly.
export JAVA_HOME=/usr/local/jdk1.3

# The line below defines a variable which will be passed as a "-cp"
argument
# to Java (i.e. the CLASSPATH).
CP=$SERVLET_CLASSPATH:$JSP_COMPILER

# The following line invokes the JSP compiler,
echo Building JSP
java -cp $CP org.apache.jasper.JspC -d $WEB_DIR -uriroot $WEB_DIR $WEB_DIR/
*.jsp
```

4.3.2 Web Archive creation script, `mkwar`

Once the WebBot Web Application has been compiled by the previous scripts is must be built into a Web Archive (WAR). The `mkwar` script does this and is shown below.

```
#!/bin/sh

source ./webapp.env
# Remove old web app
rm -rf $WEBAPP_NAME


# Create new webapp directory and copy in classes and libraries into
# webapp's
# WEB-INF directory.
mkdir $WEBAPP_NAME
cp -a web/WEB-INF $WEBAPP_NAME
mkdir $WEBAPP_NAME/WEB-INF/classes
cp src/*.class $WEBAPP_NAME/WEB-INF/classes
cp src/*.ini $WEBAPP_NAME/WEB-INF/classes
cp web/*.class $WEBAPP_NAME/WEB-INF/classes
cp -a lib $WEBAPP_NAME/WEB-INF

# Copy static content into webapp's root directory
cp web/*.txt $WEBAPP_NAME/
cp -a web/images $WEBAPP_NAME


# build webapp web archive (WAR) file
cd $WEBAPP_NAME
jar -cf ../$WEBAPP_NAME.war *
```

4.3.3 Install script, `installit.sh`

For your convenience there is an install script provided which builds and installs the Web Application. This script simply calls the previous `mkwebapp` and `mkwar` scripts before copying the WAR file into the Servlet Container's webapps directory. The script is shown below.

```
#!/bin/sh
source ./servlet.env
source ./webapp.env
./mkwebapp
./mkwar
# Remove old webapplication
rm -rf $SERVLET_CONTAINER_WEBAPPS/$WEBAPP_NAME
cp $WEBAPP_NAME.war $SERVLET_CONTAINER_WEBAPPS
```

4.3.4 Start script, `start.sh`

The start script should be executed in order to start the Servlet Container (along with any WebBot Web Applications you have installed into it). The script provided is specific to the Tomcat Servlet Container, and will need to be modified if you are using a different Servlet Container. The start script is reproduced below.

```
#!/bin/sh
export JAVA_HOME=/usr/local/jdk1.3.1_01
export CATALINA_HOME=/misc/jakarta-tomcat-4.0.4
$CATALINA_HOME/bin/startup.sh
```

4.3.5 Stop script, `stop.sh`

Use the stop script when you wish to shutdown the servlet container. For example, this is usually necessary when deploying a new version of a web application. Again this script is specific to the Tomcat Servlet Container. The script will require modification if your are using a different Servlet Container. The stop script is shown below.

```
#!/bin/sh
export JAVA_HOME=/usr/local/jdk1.3.1_01
export CATALINA_HOME=/misc/jakarta-tomcat-4.0.4
$CATALINA_HOME/bin/shutdown.sh
```

This completes the definition of the build, install and run scripts for the tutorial. They provide a good starting point for developing your own WebBot applications. The section, *Configuring the Servlet Container*, deals with the configuration files required by the Servlet Container. If you have followed the installation instructions up to this point, then the tutorial example should be ready to run. The next section will take you through the process of compiling and running the example. Spend a few minutes trying out the calculator; having done so, you will be better placed to understand the subsequent sections that discuss how its functionality is implemented.

4.4 Compiling and running the application

To compile and install the application, run the `installit.sh` script, as shown below:

```
cd $JACK_HOME/aos/jack/examples/calculator/
./installit.sh
```

To run the Servlet Container follow the steps below:

```
cd $JACK_HOME/aos/jack/examples/calculator/
./start.sh
```

Now run your browser (if it's not already running that is!) and point it to the URI: `http://localhost:8080/calculator`. This should bring up the calculator web page. Type a couple of integers into the text fields and then click on one of the buttons (e.g. *Add*). The result will appear in the Results Pane. To generate an error string, enter a zero into the righthand text field, and click on the *Divide* button.

You are welcome to explore the calculator to your heart's content, but you will quickly become bored. When you have finished, kill the Servlet Container by running the stop.sh script as shown below:

```
cd $JACK_HOME/aos/jack/examples/calculator/
./stop.sh
```

4.5 Configuring the Servlet Container

To configure the Servlet Container to handle our application, a `web.xml` must be created. Other configuration may be required for your particular Servlet Container, so please refer to your Servlet Container's documentation. The Tomcat 4.x Servlet Container should not require any configuration changes other than the WebBot Web Application's `web.xml` file.

4.5.1 Overview of `web.xml`

You should be able to leave most of the parameters in a default `web.xml` unchanged. Hence, the easiest approach to developing a new WebBot application is to take a copy of the WebBot version of `web.xml`, and only edit the parameters explained in this section. Of course, your particular application may require that you alter parameters not covered in this section; that being the case, your Servlet Container's documentation should be your first port of call.

The `web.xml` configuration file defines the *servlets* that make up your application. The tutorial example only has one servlet, but you can create an application containing a number of servlets.

4.5.1.1 Parameters used by WebBot

The file contains two major definitional elements. The `<servlet>` definition is by far the larger, and contains the servlet's initialisation parameters. The `servlet-mapping` defines the mapping between the URL patterns and servlets, i.e. which servlet to run, given a particular URL. The structure and function of the `<servlet>` and `servlet-mapping` elements is outlined below.

- servlet: Each servlet definition is delimited by "`<servlet>`" and "`</servlet>`". The key parameters of the servlet definition are as follows:

  – servlet-name: A unique identifier of the servlet.

    ```
    <servlet-name>calculator</servlet-name>
    ```

  – servlet-class: The class that implements the servlet. This class is part of the JACK distribution and will be included in the `jack.jar` file which comes with that distribution.

    ```
    <servlet-class>aos.web.webbot.portal.WebPortal</servlet-class>
    ```

The remaining parameters (`<init-param>`) are specific to WebBot, i.e. they are initialisation parameters that pertain to WebBot, rather than the Servlet Container per se. Each initialisation parameter is bounded by `<init-param>` and `</init-param>`. For example:

```
<init-param>
  <param-name>type</param-name>
  <param-value>calculator.DispatcherAgent</param-value>
</init-param>
```

The WebBot initialisation parameters are as follows:

– The class name of the request-handling JACK agent. WebBot uses this to create an agent to handle the incoming requests.

```
<init-param>
  <param-name>type</param-name>
  <param-value>DispatcherAgent</param-value>
</init-param>
```

– The instance name of the request handling agent. WebBot uses this to create and locate the request-handling agent.

```
<init-param>
  <param-name>name</param-name>
  <param-value>requestHandler</param-value>
</init-param>
```

– The class name of the class to invoke when the WebPortal cannot find an agent to handle the request. In this tutorial, this class is defined as a side effect of compilation of the JSP file, `NoHandlerErrorPage.jsp`.

```
<init-param>
  <param-name>nohandler</param-name>
  <param-value>NoHandlerErrorPage</param-value>
</init-param>
```

– The class name of the class to invoke when the agent handling the request fails (in the JACK Agent Language sense of the term "fail") to complete its processing of the event. In this tutorial, this class is defined as a side effect of compilation of the JSP file, `NoServiceErrorPage.jsp`.

```
<init-param>
  <param-name>noservice</param-name>
  <param-value>NoServiceErrorPage</param-value>
</init-param>
```

- servlet-mapping: Each servlet mapping definition is delimited by "`<servlet-mapping>`" and "`</servlet-mapping>`". The tutorial example has two mappings, one for JSP pages and one for HTML form handling. The servlet mapping maps each `servlet-name` to the `url-pattern` that invokes it. Thus, the servlet defined above (i.e. `calculator`) will handle all requests which end in `.jsp` or `.webform`. As you will see later, the URL pattern is used to trigger the appropriate JACK plan.

```
<servlet-mapping>
  <servlet-name>calculator</servlet-name>
  <url-pattern>*.jsp</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>calculator</servlet-name>
  <url-pattern>*.webform</url-pattern>
</servlet-mapping>
```

4.6 Creating a JSP file

As explained in section *JavaServer Pages (JSP)* in the JACK WebBot Architecture chapter, by far the easiest way to create web pages with WebBot is to use Java Server Pages (JSP) . This tutorial example adopts this approach. The class file compiled from, `$WEBAPP_DIR/web/calculator.java`, is used to generate a web page when the server gets a URI of the form `*.jsp`. This file is generated from the file `calculator.jsp`. If you are familiar with HTML, then, having seen the screen snapshot in the figure above, you will be expecting the `.jsp` file to reference a collection of `.gif` files. Indeed, the image is made up of many small `.gif` files organised into table cells. To reduce the clutter in this `.jsp` file, the table rows containing the `.gif` references are inserted using *include* directives. The include files have names of the form "`tr-graphics-*.txt`". Despite this, there remains a fair amount of table data cell clutter (bounded by `<td>` and `</td>`). To further improve the readability of `calculator.jsp`, such table data cells are inserted via *include* directives of files named "`td-cells-*.txt`".

The file, `calculator.jsp`, is included below. Although the file has been formatted to improve its readability, HTML can be hard to read at the best of times. Essentially the file consists of a few lines of HTML header information, followed by the retrieval of some string values from the WebBot agent; this is followed by an HTML `form` whose input fields are spread across a number of HTML table cells. The final part of the table contains the rows that hold the "Results" and "Errors" strings (retrieved by the `getValue` scriptlets at the start of the file). The file ends with a few HTML syntactic necessities.

To make full use of WebBot you will need to have a solid understanding of *interactive forms* in HTML. To understand the example below, you only need to know a few things about HTML forms:

- The form is sent to the server using an *HTTP GET request*. The data specified in the form is appended to the URI found in the "ACTION" attribute of the form, and is prefixed by a "?". For example, if the server URI is "`localhost:8080`", the "ACTION" attribute is "`/calculator/form.webform`", and the form data contains two values "`value1=5`" and "`value2=6`", then the URI would be: "`http://localhost:8080/calculator/form.webform?value1=5&value2=6`".

- The form is made of a collection of "INPUT" elements, known as "form controls". The "TYPE" attribute defines what sort of input field the INPUT element will be (e.g. "TEXT"). The buttons in our example are images, thus their type is "`type=image`". INPUT elements of this type have an associated "ACTION" that defines what should happen if the user clicks on the image. In our example, the ACTION is to submit the form to the server, i.e. "`action=submit`".

  - Each INPUT element also has a "NAME"; this is used to identify the parameters submitted in the form (`operand1` and `operand2` are examples of two named parameters in `calculator.jsp`).

  - One curiosity to be aware of is that INPUT elements of type "image" will submit *two* parameter/value pairs. The first parameter is of the form "*name*.x", and the second "*name*.y". The values of these parameters are the x and y window coordinates of the mouse pointer when the mouse button was pressed. These coordinates are not used in this tutorial example. All the JACK agent cares about is which button was pressed; thus, a simple test to see if *name*.x is non-null will suffice to select the operator to be applied to the two operands.

You should now be ready to cast your beady eye over `calculator.jsp`, shown below. To help you focus on the key sections, they are emboldened.

```
<html>

<head>
<meta http-equiv="Content-Type" content="text/html;
     charset=iso-8859-1">
</head>

<body bgcolor=#ff3300 marginwidth=0 marginheight=0 topmargin=0
     leftmargin=0>

<!-- Assignment of response values from the agent.
    These response values form part of the web page which is
    presented to the user. -->
<% String previousResult =
(String)request.getSession().getValue("previousResult"); %>
<% String currentResult =
(String)request.getSession().getValue("currentResult"); %>
<% String previousError =
```

```
(String)request.getSession().getValue("previousError"); %>
<% String currentError =
(String)request.getSession().getValue("currentError"); %>

<!-- When the user clicks on one of the submit buttons,
     the parameter/value pairs in the form are sent to the URI
     specified in the forms "action",
     i.e. "/calculator/form.webform". -->
<form method="GET" action="/calculator/form.webform"}

<!-- To reduce the clutter in this jsp file, the table rows
     containing the gifs, which make up the image presented
     to the user, are inserted using "include" directives.
     The include files are named "tr-graphics-*.txt".
     This still leaves a fair amount of table data cell
     clutter (bounded by <td> and </td>). To improve
     the readability of this file, such table data cells
     are inserted via "include" directives of files named
     "td-cells-*.txt". -->

<table cellspacing="0" cellpadding="0" border="0" >
  <%@ include file="tr-graphics-1.txt" %>
  <tr>
    <%@ include file="td-cells-1.txt" %>

    <!-- This is the input element for the "Add" button. -->
    <td width=85 height=60 valign="top" colspan=5 rowspan=9>
      <input type="image"
            action=submit
            name="add"
            alt="Add the two operands"
            src="images/button13_2.gif"
            width=85 height=60 align="top" border=0>
    </td>
    <%@ include file="td-cells-2.txt" %>

    <!-- This is the input element for the "Subtract" button. -->
    <td width=85 height=60 valign="top" colspan=4 rowspan=9>
      <input type="image"
            action=submit
            name="subtract"
            alt="Subtract the two operands"
            src="images/button29_2.gif"
            width=85 height=60 align="top" border=0>
    </td>
    <%@ include file="td-cells-3.txt" %>

    <!-- This is the input element for the "Multiply" button. -->
    <td width=84 height=60 valign="top" colspan=5 rowspan=9>
      <input type="image"
            action=submit
            name="multiply"
            alt="Multiply the two operands"
            src="images/button321_2.gif"
            width=84 height=60 align="top" border=0>
    </td>
    <%@ include file="td-cells-4.txt" %>

    <!-- This is the input element for the "Divide" button. -->
    <td width=84 height=60 valign="top" colspan=5 rowspan=9>
      <input type="image"
```

```
                  action=submit
                  name="divide"
                  alt="Divide the two operands"
                  src="images/button427_2.gif"
                  width=84 height=60 align="top" border=0>
     </td>
     <%@ include file="td-cells-5.txt" %>
</tr>
<%@ include file="tr-graphics-2.txt" %>

<!-- The two input fields, below, hold the operand values
     to be sent to the agent. -->
<tr>
    <%@ include file="td-cells-6.txt" %>
    <td width=84 height=41 valign="top" colspan=7 rowspan=2
       bgcolor="#FFFFFF">
      <div align="center">
        <input type="text" name="operand1" size="5">
      </div>
    </td>
    <%@ include file="td-cells-7.txt" %>
</tr>
<tr>
    <%@ include file="td-cells-8.txt" %>
    <td width=84 height=41 valign="top" colspan=6 rowspan=2
       bgcolor=#FFFFFF>
      <div align="center">
        <input type="text" name="operand2" size="5">
      </div>
    </td>
    <%@ include file="td-cells-9.txt" %>
</tr>

<%@ include file="tr-graphics-3.txt" %>

<!-- The parameters, previousResult and currentResult,
     returned by the agent are inserted in this row. -->
<tr>
    <%@ include file="td-cells-10.txt" %>
    <td width=443 height=46 valign="top" colspan=31 rowspan=2
       bgcolor=#FFFFFF>
      <p><%=previousResult%><br>
         <%=currentResult%></p>
    </td>
    <%@ include file="td-cells-11.txt" %>
</tr>

<%@ include file="tr-graphics-4.txt" %>

<!-- The parameters, previousError and currentError,
     returned by the agent are inserted in this row. -->
<tr>
    <%@ include file="td-cells-12.txt" %>
    <td width=443 height=46 valign="top" colspan=31
       bgcolor=#FFFFFF>
     <p><%=previousError%><br>
        <%=currentError%></p>
    </td>
    <%@ include file="td-cells-13.txt" %>
</tr>
```

```
   <%@ include file="tr-graphics-5.txt" %>
```

```
</table>
</form>
</body>
</html>
```

So, if the user enters "4" into the left text field and "2" into the right field, and clicks on the "Multiply" button, then the URI submitted to the server will be as follows (assuming that the x and y coordinates are 31 and 34 respectively):

```
http://localhost:8080/calculator/
form?operand1=4&operand2=34&multiply.x=31&multiply.y=34
```

4.7 Defining DispatcherAgent agent

The request-handling JACK agent is fully defined in the file:

```
$WEBAPP_DIR/src/DispatcherAgent.agent
```

The file looks much like any other `.agent` file: it defines an agent, the events handled, the plans used, and a private beliefset. However, if you look closely, you will notice a number of differences with respect to the `.agent` files you have seen in the past:

- It imports a number of `javax.servlet`, `aos.web.webbot.portal` and `aos.web.webbot.session` classes.

- The `DispatcherAgent` agent extends `WebSessionAgent` and implements `WebRequestHandler`.

- It handles `WebSessionRequest` and `WebDispatch` events (the latter is an extension of the WebBot `WebRequest` event).

Before looking at the constituent parts of `DispatcherAgent.agent` in more detail, it is worth briefly restating the functionality expected of this agent. The `DispatcherAgent` agent implements a very simple four-function calculator. All of the calculator functions are handled by a single JACK plan (`FormResponse`). The `DispatcherAgent` agent applies an arithmetic operator to two integer operands and returns the result. If the operands are valid arguments to the chosen arithmetic operator, then the result is displayed in the middle pane (which will also contain the result of the previous computation, if any). However, if the operands are invalid arguments to the selected operator, then an error message is returned.

The agent needs to maintain a history of the previous two results and error messages (if any) so that these can be returned and displayed on the web page. Note that the agent does not check the types of the operands. If the operands are non-integers, the agent will simply fail to service the request. This leads Tomcat to bring up a web page indicating that there is no such service, rather than specifically flagging it as a type violation. Recall that this page (`NoServiceErrorPage.jsp`) is referenced in `web.xml` via the `noservice` parameter.

The nature of the interaction that any request-handling agent (such as `DispatcherAgent`) needs to have with the client is as follows. At its simplest, the agent receives an incoming `WebRequest`. This event's signature will either trigger one of the agent's session-selection plans, or fail to match any of its plans. In the latter case, and in fact if there is a failure of any kind, it will result in the web page specified in the `noservice` parameter in `web.xml`.

The session-selection plan is supposed to determine which `WebSessionAgent` to dispatch a `WebSessionRequest` to. The `WebSessionRequest` is created by invoking the agent's `createSessionRequest(WebRequest event)` method.

The `WebSessionRequest` so created will then trigger one of the session agent's plans according to the normal JACK Agent Language rules. This sets off a computational sequence culminating in the assignment of variables that will be part of the response page returned to the client. More specifically, the agent's `putValue` method will store the named parameters' values in the response object, and will also set the value of the response page (e.g. `calculator/calculator.jsp`).

4.7.1 Imported classes

This tutorial example could hardly be simpler in terms of its use of WebBot, and as a result, the imported classes are the very minimum required to build a WebBot application.

- The `HttpServletRequest` and `HttpServletResponse` provide the HTTP-specific request and response functionality of the Servlet API:

  ```
  import javax.servlet.http.HttpServletRequest;
  import javax.servlet.http.HttpServletResponse;
  ```

- Having used the JACK Agent Language, you will be thoroughly familiar with the next line - it imports all of the JACK Agent Language functionality:

  ```
  import aos.jak.agent.Agent;
  ```

- The `aos.web.webbot.portal` package provides `HttpServlet` functionality. The `WebPortal` class, `aos.web.webbot.portal.WebPortal`, is a servlet that implements the `javax.servlet.Servlet` interface by extending the class `javax.servlet.http.HttpServlet`. The servlet is configured with the parameters `type` and `name` in the file `web.xml`. These parameters specify the class and instance name of the agent that handles incoming requests. When an HTTP request is received, if there is no agent of the specified `name`, the `WebPortal` creates one of the specified `type` with the specified `name` (specified in `web.xml`).

- The `aos.web.webbot.session` package provides the `HTTPSession` functionality and provides methods for handling `WebSessionRequest`s, and other session-related activities.

4.7.2 The agent definition

The `DispatcherAgent` agent handles incoming requests. It *extends* `WebSessionAgent` and *implements* `WebRequestHandler`. In general, your WebBot application will define such an incoming-request-handling agent. This agent can then either handle the request fully, or dispatch the request to one of a number of session-handling agents.

The agent definition is shown below. The events and plans will be explained in the following two sections. The beliefset is initialised using the contents of the file `history.ini`.

```
public agent DispatcherAgent extends WebSessionAgent
                             implements WebRequestHandler {
    #handles event WebDispatch;
    #posts event WebDispatch dispatch;
    #handles event WebSessionRequest;
    #uses plan SelectSession;
    #uses plan FormResponse;
    #uses plan SimpleJSPResponse;
    #private data History history("history.ini");

    public boolean handle(WebPortal p,
                          HttpServletRequest q,
                          HttpServletResponse r) {
        System.err.println("WebSession agent called");
        System.out.println("WebSession agent called");
        WebDispatch wd = dispatch.select(p, q, r);

        if (!postEventAndWait(wd) {
            System.err.println("Failed to handle session select");
            return false;
        }

        if (wd.a == null || wd.e == null) {
          System.err.println("Got null agent or event back");
          return false;
        }

        return wd.a.postEventAndWait(wd.e);
    }

    public DispatcherAgent(String name) {
      super(name);
    }
}
```

Apart from its constructor that invokes the `WebSessionAgent` constructor, `DispatcherAgent` defines a single method, `handle(WebPortal p, HttpServletRequest q, HttpServletResponse r)`.

4.7.3 Events handled

`DispatcherAgent` handles two types of event, the standard WebBot `WebSessionRequest` event, and the `WebDispatch` event (defined below). The `WebRequest.setup method` performs various essential setup operations internal to WebBot.

```
public event WebDispatch extends WebRequest {

  public WebSessionAgent a;
  public Event           e;

  #posted as
  select(WebPortal p, HttpServletRequest q, HttpServletResponse r) {
    setup(p, q, r);
  }
}
```

4.7.4 BeliefSet

The beliefset holds 4 elements that will be accessed using the `parameter` field. The four parameters are:

1. `result` - the current result;

2. `prevResult` - the previous result;

3. `error` - the current error; and

4. `prevError` - the previous error.

These are all initialised to the empty string. The beliefset definition is shown below.

```
  beliefset History extends OpenWorld {
  #key field String parameter;
  #value field String value;
  #indexed query get(String parameter, logical String value);
}
```

4.7.5 Plans

`DispatcherAgent` has three plans: `SelectSession`, `SimpleJSPResponse` and `FormResponse`.

4.7.5.1 SelectSession Plan

The `SelectSession` plan is responsible for determining which `WebSessionAgent` to dispatch a `WebSessionRequest` to (using the method `findAgent` shown below). In this example, `findAgent` merely returns the agent itself (i.e. `DispatcherAgent`). A more typical example would select the agent appropriate to the session in question (see chapter "Tutorial Example: Multiple Sessions"). This `SelectSession` plan also creates a `WebSessionRequest` by invoking the agent's `createSessionRequest(WebRequest event)` method. Note that both the agent and the `WebSessionRequest` are returned as data of the invoking event.

```
plan SelectSession extends Plan {

    #handles event WebDispatch ev;

    body() {
      ev.a = findAgent();
      ev.e = ev.a.createSessionRequest(ev);
    }

    /**
     * Determine which agent should deal with the request.
     * If sessions are used, then it would find the
     * appropriate session agent.
     */

    #uses interface DispatcherAgent a;

    WebSessionAgent findAgent() {
      return a;
    }
}
```

4.7.5.2 FormResponse plan

The `FormResponse` plan implements the functionality of the four-function calculator. Because all of the functionality has been packed into a single plan, the plan body is quite large. To help you focus on the sections that pertain to WebBot features, those parts are emboldened. This plan handles `WebSessionRequest` events and uses the `WebSessionAgent` interface. The plan is relevant if the `orig.webapp_path` of the event is equal to "`/form.webform`". This comes from the "`action`" of the "`form`" that the calculator web page submits. `orig.webapp_path` is a convenience provided by WebBot. It is constructed by concatenating the PathInfo and ServletPath variables from the original request. The resultant `orig.webapp_path` variable is therefore guaranteed to hold all of the request URL after the servlet name. So for example for the calculator servlet the request `http://localhost:8080/calculator/form.webform` will have an orig.webapp_path of "/form.webform". It is recommended that `orig.webapp_path` path is used rather than `orig.path` or `orig.servlet_path` as their values depend on servlet mappings and are defined within `web.xml`.

With regard to WebBot, there are three important activities in the plan's body:

1. The test for which calculator button was pressed. As was explained in section *Creating a JSP File*, because the buttons were implemented using an HTML form element of `type="image"`, two parameters are sent in the URI generated by the `form`. They are of the form "*name*.x" and "*name*.y, where *name* can be "add", "subtract", "multiply" or "divide". The "x" and "y" portions of the parameters are not used - they are the window coordinates representing where the mouse pointer was when the mouse button was pressed. Because only one button can have been pressed for any given URI submission, it is only necessary to test the parameters to find one which is non-null. Of course, if "*name*.y" is non-null, then so is "*name*.x", thus it is only necessary to test for one or the other. This is done in the conditional statement below (e.g. `ev.getParameter("add.x") != null`).

2. The storing of results in the parameters to be returned to the client, by invoking the agent's `putValue` method with the parameter name and value as arguments (e.g. `me.putValue("previousError", prevError)`).

3. The nomination of the response page using the agent's `response` method (e.g. `me.response(ev, "calculator.jsp")`).

```
plan FormResponse extends Plan {

    #handles event WebSessionRequest ev;
    #uses interface WebSessionAgent me;
    #reads data History history;


    static boolean relevant(WebSessionRequest ev) {
            return ev.orig.webapp_path.equals("/form.webform");
    }


    body() {

        logical String result;
        logical String prevResult;
        logical String prevError;
        String operand1;
        String operand2;
        String newResult;

        history.get("result", result);
        history.get("error", prevError);
        operand1 = (String)ev.getParameter("operand1");
        operand2 = (String)ev.getParameter("operand2");

        if (ev.getParameter("add.x") != null) {
            newResult = operand1 + " + " + operand2 + " = " +
                        (Integer.parseInt(operand1) +
                        Integer.parseInt(operand2));
            history.add("result", newResult);
            history.add("prevResult", result.getValue());
        }
        if (ev.getParameter("subtract.x") != null) {
            newResult = operand1 + " - " + operand2 + " = " +
                        (Integer.parseInt(operand1) -
                        Integer.parseInt(operand2));
            history.add("result", newResult);
            history.add("prevResult", result.getValue());
        }
        if (ev.getParameter("multiply.x") != null) {
            newResult = operand1 + " * " + operand2 + " = " +
                        (Integer.parseInt(operand1) *
                        Integer.parseInt(operand2));
            history.add("result", newResult);
            history.add("prevResult", result.getValue());
        }
        if (ev.getParameter("divide.x") != null) {
            if (Integer.parseInt(operand2) != 0) {
                newResult = operand1 + " / " + operand2 + " = " +
                    (Integer.parseInt(operand1) /
                    Integer.parseInt(operand2));
                history.add("result", newResult);
                history.add("prevResult", result.getValue());
```

```
        }
        else {
            newResult = "Zero divide error: " + operand1 +
                        " / " + operand2;
            history.add("error", newResult);
            history.add("prevError", prevError.getValue());
        }
    }

    logical String outputResult;
    logical String error;
    logical String oldError;

    history.get("result", outputResult);
    history.get("prevResult", prevResult);
    history.get("error", error);
    history.get("prevError", oldError);

    try {
        me.putValue("currentResult", outputResult.getValue());
        me.putValue("previousResult", prevResult.getValue());
        me.putValue("currentError", error.getValue());
        me.putValue("previousError", oldError.getValue());
        me.response(ev, "calculator.jsp");
    }
    catch (Exception e) {
        System.err.println("Could not find response to: " +
                            ev.orig.webapp_path +
                            " exception:"+e);
        e.printStackTrace();

        false;
    }
  }
}
```

The only other code fragment of interest is the *Exception* that is triggered when the `try` statement fails. This can occur, for example, if the path to the JSP page specified in the `response` is incorrect. In a deployed application, we would probably want to generate a user-friendly error dialogue of some kind.

4.7.5.3 SimpleJSPResponse plan

The `SimpleJSPResponse` plan is a cut-down version of `FormResponse`. It is triggered when the page is loaded, and merely returns the current values of the four beliefset elements, `result`, `prevResult`, `error` and `prevError`.

```
plan SimpleJSPResponse extends Plan {

    #handles event WebSessionRequest ev;
    #uses interface WebSessionAgent me;
    #reads data History history;

    body() {
        logical String result;
        logical String prevResult;
        logical String error;
        logical String prevError;
        try {
            history.get("result", result);
            history.get("prevResult", prevResult);
            history.get("error", error);
            history.get("prevError", prevError);
            me.putValue("currentResult", result.getValue());
            me.putValue("previousResult", prevResult.getValue());
            me.putValue("currentError", error.getValue());
            me.putValue("previousError", prevError.getValue());
            me.response(ev, ev.orig.webapp_path);
        }
        catch (Exception e) {
            System.err.println("Could not find JSP path: " +
                            ev.orig.webapp_path);
            false;
        }
    }
}
```

4.8 Summary

This chapter presented a simple application of WebBot that combines the properties of `WebRequestHandler` with those of `WebSessionAgent` into a single agent, `DispatcherAgent`. This agent handles incoming `WebRequest` events and uses the `createSessionRequest` method to create a `WebSessionRequest` events. This event triggers a response from the session agent. Because the agent combines session-selection functionality with that of a session agent, the `WebSessionRequest` is actually handled by the same agent, `DispatcherAgent`. The session agent (`DispatcherAgent`) sets various result parameters in the response object and returns a reference to the response page, `calculator.jsp`.

The next chapter will augment this example so that the dispatch functions and session responses are handled by separate agents.

5 Tutorial example: multiple sessions

5.1 Introduction

The simple four-function calculator example illustrated how to implement an agent that responds to incoming HTTP requests. If you were to deploy the application, you would quickly discover that it does not discriminate between users. The calculator maintains a single history of results; all users would see the same history. This is unsatisfactory because each user should see their own unique calculator interaction history, not the merged history of all of the users' calculations. Indeed, the user might have two browser windows open, each with a separate history of arithmetic calculations. The server must maintain a unique interaction thread for each window; this functionality is provided by *sessions*.

Sessions form part of the Servlet API. WebBot provides support for the implementation of sessions through the provision of the `WebSessionAgent` and `WebSessionRequest` classes.

Although the Servlet API provides support for tracking sessions through the use of *cookies*, this method is fairly limited. For example, if the client session occurred on a machine which is shared by others (e.g. in an internet cafe environment), then it is feasible for a new user to come along and pick up the previous user's session. Another drawback of cookies is that they restrict you to one session per browser, i.e. if the user tries to have two browser windows open accessing the servlet, they will map to the same session.

A safer method of tracking sessions is to use URL rewriting. In this approach, a session identifier is stored in a hidden attribute on the web page. This is then written into the URL before sending the HTTP request back to the server. This is the preferred method when using WebBot. If you store the session id in the attribute "`jsessionid`", and arrange for that to be included in the URL, then the `WebPortal setup` method will extract it and store it in the parameter "`id`". `WebPortal` will also pick up the "`referrerid`" and store it in the parameter "`refid`". This can be used to determine if the web page has a referrer, and so can be used in most cases to prevent someone from dropping in on a session (unless of course, the person has written their own browser).

5.2 Overview of modifications for sessions

As alluded to in the previous section, the simple calculator example will be modified so that it can maintain a separate interaction thread for each calculator browser window instance. The `DispatcherAgent` will be broken up into two separate types of agent: one which performs the message dispatch function, and another which manages the interaction for a given session. The former will still be named `DispatcherAgent` and there will be only one such agent instance in the application. The latter will be named `SessionCalculator` and at any one time, there will be as many instances as there are sessions.

5.3 Location of the multi-session tutorial example

The multi-session example can be found in the `sessioncalc` directory under the `examples` directory. The directory structure is very similar to that of the simple calculator example (previous figure) and is shown in the figure below.
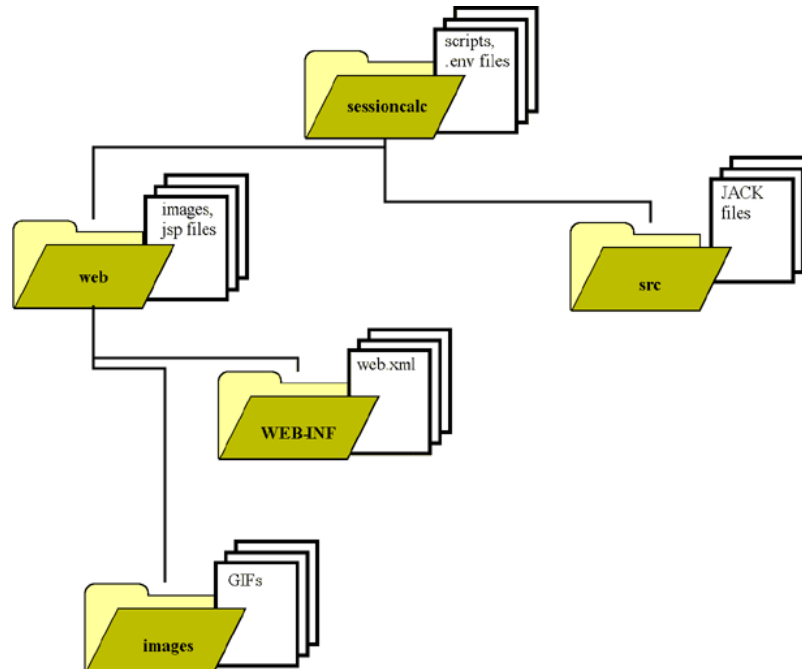

Figure 5-1: Directory Structure for the Multi-Session Calculator Example

5.4 Setting up the build and run scripts

The build and run scripts are largely unchanged from the previous simple calculator example. The webapp.env should be the only file requiring modification. The updated file is shown below.

```
# Name of WebBot web application
WEBAPP_NAME=sessioncalc

# Directory where you have installed JACK
JACK_HOME=$HOME/Agent_Software/

# Directory where you are developing your WebBot web application
WEBAPP_DIR=$JACK_HOME/aos/jack/examples/calculator
```

5.5 Configuring the Servlet Container

5.5.1 Changes to the `web.xml` file

The file `web.xml` requires that the `<servlet-name>` be altered to reference the multi-session version of the calculator:

`<servlet-name>`**sessioncalc**`</servlet-name>`

The remaining parameters need to be changed so that `calculator` is replaced by `sessioncalc`;

- servlet-mapping: Each servlet mapping definition is delimited by `<servlet-mapping>` and `</servlet-mapping>`. The servlet mapping maps each `servlet-name` to the `url-pattern` that invokes it. Thus, the servlet defined above (i.e. `sessioncalc`) will handle all requests with a URL of the form `*.jsp` or `*.webform`.

```
<servlet-mapping>
  <servlet-name>sessioncalc</servlet-name>
  <url-pattern>*.jsp</url-pattern>
</servlet-mapping>

<servlet-mapping>
  <servlet-name>sessioncalc</servlet-name>
  <url-pattern>*.webform</url-pattern>
</servlet-mapping>
```

- The following parameter has been added so that the application can determine the type of the session agent.

```
<init-param><param-name>session-agent-type
          <!-- The type of the session handling agent. -->
          </param-name>
          <param-value>SessionCalculator
          </param-value>
</init-param>
```

This completes the definition of the build and run scripts for the multi-session calculator example.

5.6 Compiling and running the application

To compile and install the application, run the `mkwebapp` script, as shown below:

```
cd $JACK_HOME/aos/jack/examples/calculator/
./installit.sh
```

To run the Servlet Container follow the steps below:

```
cd $JACK_HOME/aos/jack/examples/calculator/
./start.sh
```

Now run your browser (if it's not already running that is!) and point it to the URI: `http://localhost:8080/sessioncalc`. This should bring up the calculator web page. Type a couple of integers into the text fields and then click on one of the buttons (e.g. *Add*). The result will appear in the Results Pane. To generate an error string, enter a zero into the righthand text field, and click on the *Divide* button.

Open another browser window and point it to the same URI (i.e. `http://localhost:8080/sessioncalc`). Perform some different calculations in each browser window and verify that each window displays a separate history in the Results Pane and/or the Errors Pane. When you have finished, kill the Servlet Container by running the `stop.sh` script.

5.7 Creating a JSP file

The easiest way to create web pages with WebBot is to use Java Server Pages (JSP) . The file, `$WEBAPP_DIR/web/calculator.java`, is used to generate a web page when the server gets a URI of the form `sessioncalc/*.jsp` or `sessioncalc/*.webform`. This file is generated from the file `calculator.jsp`. In keeping with the approach adopted for the calculator example in the chapter titled "Tutorial Example: Simple Calculator", the use of *include* directives reduces the clutter in `calculator.jsp`. The include files have names of the form "`tr-graphics-*.txt`" and "`td-cells-*.txt`".

The file, `calculator.jsp`, is very similar to the file, `calculator.jsp`, from the "Tutorial Example: Simple Calculator" chapter. The changes pertain to the use of sessions , and are shown below.

Recall that the main JSP file, `calculator.jsp`, will compile into the file `calculator.java`. In order to make use of the WebBot session facilities, this `.java` file needs to import the `WebSessionAgent` package. The first scriptlet, below, does just that. The second scriptlet declares an instance of `WebSessionAgent`, and the third retrieves the session using `request.getsession().`

```
<!-- The WebSessionAgent, me, is assigned using
     request.getSession(). -->
<%@ page import="aos.web.webbot.session.WebSessionAgent" %>
<%! public WebSessionAgent me; %>
<% me = (WebSessionAgent) request.getSession(); %>
```

The GET action has been changed to reference sessioncalc rather than calculator, but is otherwise the same as that in calculator.jsp in the "Tutorial Example: Simple Calculator" chapter. The subsequent scriptlet invokes the method, WebSessionAgent.sessionFormMarkup(), on the WebSessionAgent instance, me.

```
<!-- When the user clicks on one of the submit buttons,
     the parameter/value pairs in the form are sent to the URI
     specified in the forms "action",
     i.e. "/sessioncalc/form.webform". -->
<form method="GET" action="/sessioncalc/form.webform">

<!-- The sessionFormMarkup method adds a "jsessionid"
     into the form. -->
<%=me.sessionFormMarkup()%>
```

As before, the form is sent to the server using an *HTTP GET request*. The data specified in the form is appended to the URI found in the "ACTION" attribute of the form, and is prefixed by a "?". The form data will include the jsessionid added in by the WebSessionAgent.sessionFormMarkup() method. For example, if the server URI is "localhost:8007", the "ACTION" attribute is "/sessioncalc/form.webform", the jsessionid is SessionCalc_1 and the form data contains two values "operand1=5" and "operand2=6", then the URI would be: "http://localhost:8080/sessioncalc/ form.webform?jsessionid=SessionCalc_1&operand1=5&operand2=6". Additionally, the attribute/values representing the button that was pressed would also be included, for example: "add.x=58&add.y=28". Recall that INPUT elements of type "image" will submit *two* parameter/value pairs. The first parameter is of the form "*name*.x", and the second "*name*.y". The values of these parameters are the x and y window coordinates of the mouse pointer when the mouse button was pressed. These coordinates are not used in this tutorial example. All the session agent needs is an indication of which button was pressed.

5.8 DispatcherAgent agent

This agent takes on the role of the *root* agent shown in Figure [give name]. The agent definition is reproduced below. In contrast to the non-sessions example in the chapter titled "Tutorial Example: Simple Calculator", this version of DispatcherAgent does not extend WebSessionAgent; this is because it is only responsible for session-selection and dispatch. Managing an individual session is the responsibility of the SessionCalculator agent.

The DispatcherAgent handles incoming WebRequests (specifically, WebDispatch events). Its SelectSession plan is used to match the request to an existing session. If the WebRequest does not match an existing session, then the plan DefaultRequestHandler creates a new session to handle it and stores a reference to that agent in its Sessions beliefset. The MonitorSession plan kills off the session agent if it has been inactive for 10 minutes or more. The handle method uses the createSessionRequest method to create a WebSessionRequest event that then gets posted using postEventAndWait.

```
public agent DispatcherAgent extends Agent implements WebRequestHandler {

    #handles event WebDispatch;
    #posts event WebDispatch dispatch;
    #handles event SessionAccess;
    #uses plan MonitorSession;
    #uses plan SelectSession;
    #uses plan DefaultRequestHandler; // The SelectSession plan
                                      // takes precedence.

    #private data Sessions sessions(); // Beliefset of all sessions.


    public boolean handle(WebPortal p,
                          HttpServletRequest q,
                          HttpServletResponse r) {
       WebDispatch wd = dispatch.select(p, q, r);
       if (!postEventAndWait(wd))
           return false;
       if (wd.a == null)
           return false;
       WebSessionRequest wsr = wd.a.createSessionRequest(wd);
       return wd.a.postEventAndWait(wsr);
    }

    public DispatcherAgent(String name) {
       super(name);
    }
}
```

5.8.1 DispatcherAgent's plans

DispatcherAgent has three plans: DefaultRequestHandler, MonitorSession and
SelectSession.

5.8.1.1 DefaultRequestHandler Plan

This plan handles requests by creating a new session agent to forward the request to. The
WebPortal.getInitParameter method returns the agent type. The agent type is specified in
the file web.xml. The agent is given a unique name of the form SessionCalc_<n>. A new
session agent is created and stored in WebDispatch.a. The session is added to the sessions
beliefset as a tuple containing the agent name and the WebSessionAgent.

```
plan DefaultRequestHandler extends Plan {

    #handles event WebDispatch ev;

    static long count = 0;

    #modifies data Sessions sessions;
    #uses interface Agent self;

    body()
    {
        WebPortal p = (WebPortal) ev.servlet;
        String type = p.getInitParameter("session-agent-type");
        String name = "SessionCalc_" + (count++);
        type != null ;
        System.err.println("Creating agent " + type +
                           " [" + name +"]");
        ev.a = (WebSessionAgent) Kernel.createAgent(type, name,
                                                    null);
        sessions.assert(name, ev.a, self.timer.getTime());
    }
}
```

5.8.1.2 MonitorSession plan

This plan is triggered when a SessionAccess event is received. It looks up the session entry for the event, ev, and waits for 10 minutes. The SelectSession plan updates the session's time entry when a WebDispatch event is received for the session. Effectively, the clock is reset every time there is activity from a session (i.e. when a new ev.id is asserted). This is because the sessions.get(ev.id, a, ev.time) statement will fail if there is a new assertion for that key. If the sessions.get statement does not fail, then the session is removed from the beliefset and the agent is destroyed via the finish() method.

```
plan MonitorSession extends Plan {

    #handles event SessionAccess ev;

    #uses data Sessions sessions;

    body()
    {
        logical WebSessionAgent $agent;
        sessions.get(ev.id, $agent, ev.time);
        WebSessionAgent a = (WebSessionAgent) $agent.getValue() ;

        @waitFor(elapsed(600));

        sessions.get(ev.id, a, ev.time);
        sessions.remove(ev.id, a, ev.time);
        a.finish();
    }

}
```

5.8.1.3 SelectSession plan

This plan determines which `WebSessionAgent` to dispatch a `WebSessionEvent` to. This is achieved by looking up the `ev.id` in the `sessions` beliefset. The session is re-asserted with a new time signature (effectively resetting the session timeout clock).

```
plan SelectSession extends Plan {

    #handles event WebDispatch ev;

    static boolean relevant(WebDispatch ev) {
        return ev.id != null ;
    }

    #uses data Sessions sessions;
    #uses interface Agent self;

    logical WebSessionAgent $agent;
    logical long $access;

    context() {
        sessions.get(ev.id, $agent, $access);
    }

    body() {
        ev.a = (WebSessionAgent) $agent.getValue();
        long time = self.timer.getTime();
        sessions.assert(ev.id, ev.a, time);
    }
}
```

5.8.2 DispatcherAgent's events

5.8.2.1 SessionAccess.event file

This event is posted when a new session is added to the sessions beliefset (see *DispatcherAgent's BeliefSet* (`Sessions.bel`)).

```
public event SessionAccess extends Event {

    public String id;
    public long time;

    #posted as
    access(String i,long t)
    {
        id = i;
        time = t;
    }
}
```

5.8.2.2 WebDispatch.event file

This event is unchanged from that presented in the "Tutorial Example: Simple Calculator" chapter. The `WebRequest.setup method` performs various essential setup operations internal to WebBot.

```
public event WebDispatch extends WebRequest {

    public WebSessionAgent      a;
    public Event                e;

    #posted as
    select(WebPortal p, HttpServletRequest q,
           HttpServletResponse r) {
      setup(p, q, r);
    }
}
```

5.8.3 DispatcherAgent's BeliefSet (Sessions.bel)

This beliefset holds all of the active sessions. When there is activity on a session, a new assertion is added to the `Sessions` beliefset by the plan, `SelectSession.plan`. This has the side effect of posting a `SessionAccess` event via the `newfact` method. A given assertion is removed when there has been no activity on its session for 10 minutes or more (`MonitorSession.plan`).

```
public beliefset Sessions extends ClosedWorld {

    #key field String id;
    #value field WebSessionAgent agent;
    #value field long access;

    #indexed query get(String i, WebSessionAgent a, long x);
    #indexed query get(String i, logical WebSessionAgent a, long x);
    #indexed query get(String i, logical WebSessionAgent a,
                       logical long x);

    #posts event SessionAccess sa;

    public void newfact(Tuple tx, BeliefState is, BeliefState was)
    {
        Sessions__Tuple t = (Sessions__Tuple) tx;
        postEvent(sa.access(t.id, t.access));
    }
}
```

5.9 SessionCalculator agent

This agent is responsible for responding to incoming requests. It implements the functionality of the four-function calculator (previously embedded within the `DispatcherAgent` in the "Tutorial Example: Simple Calculator" chapter).

```
public agent SessionCalculator extends WebSessionAgent {

    #handles event WebSessionRequest;

    #private data History history("history.ini");

    #uses plan FormResponse;
    #uses plan SimpleJSPResponse;


    public SessionCalculator(String name) {
      super(name);
    }
}
```

5.9.1 `SessionCalculator`'s plans

5.9.1.1 The plan `FormResponse`

Apart from changing the response to reference `sessioncalc/calculator.jsp`, this plan is unchanged from that presented in the chapter titled "Tutorial Example: Simple Calculator"

```
 me.response(ev, "sessioncalc/calculator.jsp");
```

5.9.1.2 SimpleJSPResponse plan

This plan is unchanged from that presented in the chapter , "Tutorial Example: Simple Calculator"

5.9.2 `SessionCalculator`'s BeliefSet

5.9.2.1 The BeliefSet `History.bel`

This beliefset definition is unchanged from that presented in the chapter , "Tutorial Example: Simple Calculator". The only difference is that it is now a part of the `SessionCalculator` agent instead of the `DispatcherAgent` that previously included the sessions functionality.

```
beliefset History extends OpenWorld {
    #key field String parameter;
    #value field String value;
    #indexed query get(String parameter, logical String value);
}
```

5.10 Summary

This chapter extended the simple calculator application from the chapter , "Tutorial Example: Simple Calculator" so that it handles multiple sessions. The `DispatcherAgent` of the previous chapter was divided into two separate agents, one implementing `WebRequestHandler`, the other extending `WebSessionAgent`. The former agent (named `DispatcherAgent`) handles incoming `WebRequest` events and uses the `createSessionRequest` method to create a `WebSessionRequest` event. The `WebSessionRequest` event triggers a response from the session agent, `SessionCalculator`. The session agent (`DispatcherAgent`) sets various result parameters in the response object and returns a reference to the response page, `calculator.jsp`.

Index

A
agent
    autonomous 11
agent-oriented programming 11
ASP 11

C
cookies 20, 53
createSessionRequest method 18

D
doGet method 19
doPost method 19

G
GET
    HTTP method 12

H
HTML form 12

I
init-param 39
installation
    of WebBot 21

J
JACK 11
JACK WebBot
    Architecture 15
    description 11
JavaServer Pages 11, 12, 40, 56
jsessionid 19, 20, 53, 57
JSP 11, 12, 13, 17, 18, 19, 21, 25, 33, 40, 56
    compilation 35, 39
    compiler 11, 15, 16

L
lookupHandler method 19

N
ns 25

P
param-name
    name 24, 39
    nohandler 24, 39
    noservice 24, 39
    type 24, 39
PHP 11
port
    IP 24
PUT
    HTTP method 12

R
request
    parameter 12
response
    parameter 12
root agent 18, 26

S
scriptlet 11, 25, 56
service method 12
servlet 12
    API 11, 12, 13, 21
    container 12, 13, 15
servlet-class 24, 38
servlet-mapping 24, 38
servlet-name 24, 38
session
    agent 18, 28
    selection plan 18
Sessions 16, 20, 53
sessions 20, 53, 56, 58, 62

T
Tomcat 14

U
URI 12
URL pattern 16
URL rewriting 20, 53

W
WAR 32
Web Application Archive 32
web.xml 16, 18, 19, 21, 22, 23, 38, 44, 45,
48, 55, 58
WebBot
description 11
WebPortal 18, 19, 20, 25, 39, 45, 53
WebSessionAgent 19, 28
WebSessionRequest event 18