Tapestry Designer's Guide

Tapestry Designer's Guide

Table of Contents

Chapter

1

Introduction: What is Tapestry?

An introduction to the Tapestry web application framework, introducing elements of Tapestry's design, goals and philosophy.

T apestry is a comprehensive web application framework, written in Java.

Tapestry is not an application server. Tapestry is a framework designed to be used inside an application server.

Tapestry is not an application. Tapestry is a framework for creating web applications.

Tapestry is not a way of using JavaServer Pages. Tapestry is an alternative to using JavaServer Pages.

Tapestry is not a scripting environment. Tapestry uses a component object model, not simple scripting, to create highly dynamic, interactive web pages.

Tapestry is based on the Java Servlet API version 2.1. It is compatible with JDK 1.2 and above. Tapestry uses a sophisticated component model to divide a web application into a hierarchy of components. Each component has specific responsibilities for rendering web pages (that is, generating a portion of an HTML page) and responding to HTML queries (such as clicking on a link, or submitting a form).

The Tapestry framework takes on virtually all of the responsibilities for managing application flow and server-side client state. This allows developers to concentrate on the business and presentation aspects of the application.

Scripting vs. Components

Most leading web application frameworks are based on some form of scripting. These frameworks (often bundled into a web or application server) include:

- Sun JavaServer Pages

- Microsoft Active Server Pages (ASP)

- Allaire ColdFusion

- PHP

- WebMacro

- FreeMarker

All of these systems are based on reading an HTML template file and performing some kind of processing on it. The processing is identified by directives … special tags in the HTML template that indicate dynamic behavior.

Each framework has a scripting language. For JavaServer Pages it is Java itself. For ASP it is Visual Basic. Most often, the directives are snippets of the scripting language inserted into the HTML.

For example, here's a snippet from a hypothetical JavaServer Page that displays part of a shopping cart.

```
<%
     String userName = (String)session.getAttribute("userName");
%>
<h1>Contents of shopping cart for <%= userName %>:</h1>
```

Most of the text is static HTML that is sent directly back to the client web browser. The bold text identifies scripting code.

The first large block is used to extract the user name from the HttpSession, a sort of per-client scratch pad (it is part of the Java Servlet API; other systems have some similar construct). The second block is used to insert the value of an expression into the HTML. Here, the expression is simply the value of the userName variable. It could be more complex, including the result of invoking a method on a Java object.

This kind of example is often touted as showing how useful and powerful scripting solutions are. In fact, it shows the very weaknesses of scripting.

First off, we have a good bit of Java code in an HTML file. This is a problem … no HTML editor is going to understand the JavaServer Pages syntax, or be able to validate that the Java code in the scripting sections is correct, or that it even compiles. Validation will be deferred until the page is viewed within the application. Any errors in the page will be shown as runtime errors. Having Java code here is unnatural … Java code should be developed exclusively inside an IDE.

In a real JavaServer Pages application I've worked on, each JSP file was 30% - 50% Java. Very little of the Java was simple presentation logic like `<%= userName %>`, most of it was larger

blocks needed to 'set up' the presentation logic. Another good chunk was concerned with looping through lists of results.

In an environment with separate creative and technical teams, nobody is very happy. The creative team is unlikely to know JSP or Java syntax. The technical team will have difficulty "instrumenting" the HTML files provided by creative team. Likewise, the two teams don't have a good common language to describe their requirements for each page.

One design goal for Tapestry is minimal impact on the HTML. Many template-oriented systems add several different directives for inserting values into the HTML, marking blocks as conditional, performing repetitions and other operations. Tapestry adds exactly one directive, and it's designed to look just like an HTML element.

A Tapestry component is identified by a `<jwc>` tag (Java Web Component).

For comparison, an equivalent Tapestry template to the previous JSP example:

```
<h1>Contents of shopping basket for
<jwc id="insertUserName"/>:</h1>
```

This defines a component named insertUserName on the page. Because we use just a single tag, `<jwc>`, for all components, it is not possible to say exactly what dynamic content will be inserted ... that is defined elsewhere. The `<jwc>` tag simple identifies where the dynamic content will go.

A portion of the page's specification file defines what insertUserName is and what it does:

```
<component>
  <id>insertUserName</id>
  <type>Insert</type>

  <bindings>
    <binding>
      <name>value</name>
      <property-path>application.userName</property-path>
    </binding>
  </bindings>
</component>
```

This identifies insertUserName as an Insert component; a component that inserts some text into the response page's HTML. It further identifies what gets inserted (the component's value) as the userName property of the application object.

If this were all that Tapestry did, it would still be a big step forward. However, Tapestry truly begins to distinguish itself when it comes to creating interactions on the page.

Interaction

Let's continue with a portion of the JSP that would allow an item to be deleted from the shopping cart. For simplicity, we'll assume that there's an object of class `LineItem` named item and that there's a servlet used for making changes to the shopping cart.

```
<tr> <td> <%= item.getProductName() %></td>
       <td> <%= item.getQuantity() %></td>
       <td>
<%  String URL = response.encodeURL("/servlet/update-cart?action=remove" +
                 "&item=" + item.getId());
%>
<a href="<%= URL %>">Remove</a> </td> </tr>
```

This clearly shows that in a JSP application, the designer is responsible for "knitting together" the pages, servlets and other elements at a very low level. By contrast, Tapestry takes care of nearly all these issues automatically:

```
<tr> <td> <jwc id="insertName"/> </td>
       <td> <jwc id="insertQuantity"/> </td>
       <td> <jwc id="removeAction">Remove</jwc> </td> </tr>
```

Because of the component object model used by Tapestry, the framework knows exactly "where on the page" the removeAction component is. It uses this information to build an appropriate URL that references the removeAction component. If the user clicks the link, the framework will return the page containing the removeAction and inform the component to perform the desired action. The removeAction component can then remove the item from the shopping cart.

In fact, under Tapestry, no user code ever has to either encode or decode a URL. This removes an entire class of errors from a web application (those URLs can be harder to assemble and parse than you might think!)

Tapestry isn't merely building the URL to a servlet for you; the whole concept of 'servlets' drops out of the web application. Tapestry is building a URL that will invoke a method on a component.

Tapestry applications act like a 'super-servlet'. There's only one servlet to configure. By contrast, even a simple JavaServer Pages application developed using Sun's Model 2 (where servlets provide control logic and JSPs are used for presenting results) can easily have dozens of servlets.

Security

Developing applications using Tapestry provides some modest security benefits.

First off, Tapestry applications are built on top of the Java Servlet API, so there are inherent benefits there. Most security intrusions against CGI programs (such as Perl) rely on sloppy code that evaluates portions of the URL in a system shell; this never happens when using the Java Servlet API.

Because the URLs created by Tapestry for processing client interaction are more strongly structured than the URLs in traditional solutions, there are fewer weaknesses to exploit. Improperly formatted URLs result in an exception response being presented to the user. Tapestry URLs are also harder to spoof, since they are very conversational; they often include a version number that changes during the conversation between client and server. An invalid version number also results in an exception response.

Where the Java Servlet API suffers is in client identification, since a session identifier is stored on the client either as an HTTP Cookie or encoded into each URL. Malicious software could acquire such an identifier and "assume" the identity of a user who has recently logged into the application. Because of the conversational nature of the Tapestry URLs it would be difficult for an automated intruder to progress through the application from that point.

Finally, Tapestry applications have a single flow of control: all incoming requests flow through a few specific methods of particular classes. This makes it easier to add additional security measures that are specific to the application.

Features

The framework, based on the component object model, provides a significant number of other features, including:

- Easy localization of applications

- Extremely robust error handling and reporting

- Highly re-usable components

- Automatic persistence of server-side client state between request cycles

- Powerful processing of HTML forms

- Strong support for load balancing and fail over

- Zero code generation

- Easy deployment

The point of Tapestry is to free the web application developer from the most tedious tasks. In many cases, the "raw plumbing" of a web application can be completely mechanized by the framework, leaving the developer to deal with more interesting challenges, such as business and presentation logic.

As Tapestry continues to develop, new features will be added. On the drawing board are:

- Support for easy cross-browser DHTML

- Improved XML support

- WAP / WML support

- Tighter database integration

- A real-time performance "Dashboard"

- Journaling / Playback

Chapter

2

Web Applications

T apestry has a very strong sense of what an application is. An application is an instance of a particular class (usually a subclass from a provided base class, such as `com.primix.tapestry.app.SimpleApplication`). An application consists of a number of pages and provides a number of services to the pages and components on those pages.

In other systems, there is no application per-se. There is some kind of 'home page' (or servlet), which is the first page seen when a client connects to the web application. There are many pages, servlets (or equivalent, in other frameworks) and interrelations between them. There is also some amount of state stored on the server, such as the user name and shopping cart. The sum total of these elements is the web application.

Tapestry imposes a small set of constraints on the developer, chiefly, that the application be organized in terms of pages and components (and for advanced designers, services).

These constraints are intended to be of minimal impact to the developer, imposing a minimal amount of structure. They create a common language that can be used between members of a team, and even between the technical and creative members of the team.

Under Tapestry a page is also very well defined: It consists of a component specification, a corresponding Java class, an HTML template, and a set of contained components.

By contrast, when using JavaServer Pages there are one or more servlets, embedded JavaBeans, a JSP file and the Java class created from the JSP file. There isn't a standard naming scheme or other way of cleanly identifying the various elements.

Interactivity in Tapestry is component based. If a component is interactive, such as an image button with a hyperlink (`<a>`), clicking on the link invokes a method on the component. All interactivity on a page is implemented by components on the page.

JavaServer Pages bases its interactivity on servlets. Interactive portions of a page must build URLs that reference these servlets. The servlets use a variety of ad-hoc methods to identify what operation is to take place when a link is clicked. Since there is no standard for any of this, different developers, even on the same project, may take widely varying approaches to solving similar

Because pages are components, they have a well-defined interface, which describes to both the framework and the developer how the page fits into the overall application.

A Tapestry application object provides a global place to store data between request cycles. It also provides logic that can be common in many pages of the application.

The application also provides services. Services are the bridge between URLs and components. Services are used to generate the URLs used by hyperlinks and form submissions. They are also responsible for interpreting the same URLs when they are later triggered from the client web browser.

Tapestry Components

Tapestry components are "black boxes" that are involved with both rendering HTML responses and responding to HTTP requests.

A Tapestry component is defined by its specification. The specification is an XML file that defines the type of the component, it parameters, the template used by the component, any components embedded within it and how they are 'wired up', and (less often) any assets used by the component.

At runtime, the specification is used to identify and instantiate a class for the component. When the page containing the component is rendered, the component will access its HTML template (whose location is defined in the specification) to find the static HTML and embedded components it will render.

Parameters and Bindings

Tapestry components are designed to work with each other, within the context of a page and application. The process of rendering a page is largely about pulling information from a source (example: the userName property of the application object) into a component (example: the insertUserName component) and doing something with it (example: writing it to the HTML response stream).

Each component has a specific set of parameters. Parameters have a name, a type and may be required or optional.

To developers experienced with Java GUIs, it may appear that Tapestry component parameters are the same as JavaBeans properties. This is not actually true. JavaBeans properties are set-and-forget; the designer sets a value for the property using a visual editor and the value is saved with the bean until it is used at runtime.

Parameters define the type of value needed, but not the actual value. This value is provided by a special object called a binding.

A binding is a source and sink of data. A component uses a binding to import or export a data value.

There are two types of bindings: static and dynamic. Static bindings are read-only; the value for the binding is specified in the component specification.

Dynamic bindings are more prevalent and useful. A dynamic binding uses a JavaBeans property name to retrieve the value when needed by the component.

In fact, dynamic bindings can use a property path. This is a series of property names separated by periods. We've already seen an example: application.userName. This is equivalent to the Java code `getApplication().getUserName()`.

Property paths can be very useful, since they can allow a binding to 'crawl' deeply through an object graph to access the value it needs. Let's change the example: now the application contains a user property, which is an object of class `User` that has a number of properties related to the real-life user (such as name, e-mail, account number, etc.).

We can still use an insert component to display the user's name; we just have to extend the property path to application.user.name, which is equivalent to `getApplication().getUser().getName()`.

Embedded Components

Under Tapestry, it is common to define new components by combining (or *aggregating* to use object-oriented terminology) existing components. The existing components are embedded in the containing component.

Each embedded component has an id (an identifying string) that must be unique within the containing component. Every non-page component is embedded inside some other component forming a hierarchy that can get quite deep (in real Tapestry applications, some pages have components nested three to five levels deep).

In some cases, a component will be referenced by its id path. This is a series of component ids separated by periods, representing a path from the page to a specific component. The same notation as a property path is used, but the information being represented is quite different.

For example, the id path border.navbar.homeLink represents the component named homeLink, embedded inside a component named navbar, embedded inside a component named border, embedded inside some page.

Tapestry components are "black boxes". They have a set of parameters that may be bound. Primitive components may not embed other components, or even have a template. Nearly all the built-in components are primitive; they are building blocks for building more complex components.

Alternately, a component may be implemented using a template and embedded components. In either case, the names, types or very existence of embedded components is private, hidden inside the containing component's "black box".

HTML Templates

Nearly all Tapestry components combine static HTML from a template with additional dynamic content (some few components are just dynamic content). Often, a Tapestry component embeds other Tapestry components. These inner components are referenced in the containing component's template.

Templates look like standard HTML files, though they are rarely complete HTML documents; usually they are snippets. In addition, a new element, `<jwc>` (for Java Web Component), is added to represent a component in the HTML template.

The `<jwc>` element has two forms:

```
<jwc id="component id">  body  </jwc>
```

or

```
<jwc id="component id"/>
```

The parser used by Tapestry is relatively forgiving about case and white space. Also, the component id can be enclosed in double quotes (as above) or single quotes.

The body listed above can be either static HTML or other Tapestry components or both. Elements in the body of a component are *wrapped* by the containing component. The containing component controls the rendering of the elements it wraps in its body. For example, the Conditional component may decide not to render its body and the Foreach component may render its body multiple times.

Not all Tapestry components should have a body. For example, the TextField component creates an `<input type=text>` form element and it makes no sense for it to contain anything else. Whether a component can have a body (and wrap other elements) is defined in the component's specification.

Tapestry includes a special component, InsertWrapped, which is used to render the body of a component. It makes it easy to create components that wrap other components.

In many scripting systems, scripting constructs are one-dimensional.... they appear at a certain point on the page and dynamically insert some content. Tapestry components are more powerful ... they have open and close tags and thus can contain a *body*. This body is made up of static HTML from the template and other Tapestry components. The outer component *wraps* the inner elements.

An example of this is the Border component in the Tapestry Tutorial. It shows how a component can wrap the entire content of a page.
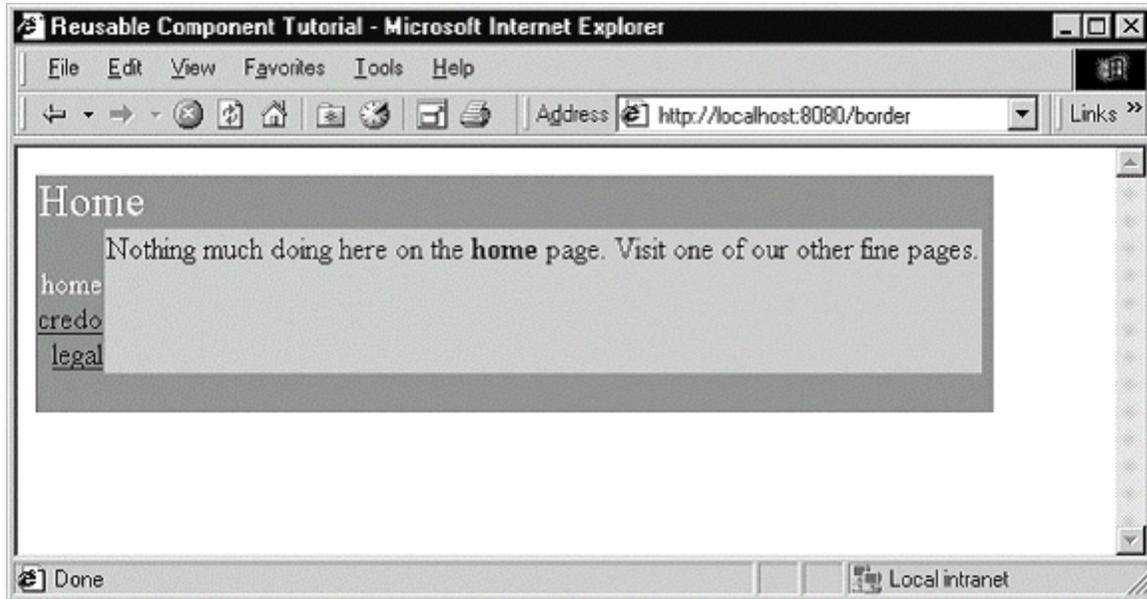
Here, the border component provides the page title and (upper left corner) and navigation controls (left side) ... all the content in the dark outer frame. The page provides the content in the light gray box in the center. In this example, the page content is simply static HTML but in a real application the content would include other components.

The Home page component has the following HTML template:

```
<jwc id="border">

Nothing  much  doing  here  on  the  <b>home</b>  page.   Visit  one  of  our  other
fine
pages.

</jwc>
```

In other words, the border component gets the produce HTML first. At some point, the page's content (wrapped inside the border component) will be produced. Afterwards, the border component will have an opportunity to produce more HTML.

The Border component has a much larger HTML template:

```
<HTML>
<head>
<title>Reusable Component Tutorial</title>
</head>
<body>
<table border=0 bgcolor=gray cellspacing=0>
  <tr valign=top>
    <td colspan=3 align=left>
      <font size=5 color="White"><jwc id="insertPageTitle"/></font>
    </td>
  </tr>
```

```
  <tr valign=top>
    <td align=right>
      <font color=white>
<jwc id="e">
        <br><jwc id="link"><jwc id="insertName"/></jwc>
</jwc>
      </font>
    </td>
    <td valign=top bgcolor=silver>
      <jwc id="wrapped"/>
    </td>
    <td> </td>
  </tr>
  <tr>
    <td colspan=3> </td>
  </tr>
</table>
</body>
</HTML>
```

The Border contains five components:

- insertPageTitle

- e

- link

- insertName

- wrapped

The e component wraps some static HTML and the link component. The link component wraps the insertName component.

When it comes time to render, Tapestry parses the HTML template and breaks them into blocks of static HTML, component starts (the `<jwc>` tag) and component ends (the `</jwc>` tag).

It figures out what each component wraps. For components with their own templates, such as Home and Border, it figures out what the *outer elements* are, the outermost static HTML and components that aren't wrapped by other components.

This information turns into a data structure:
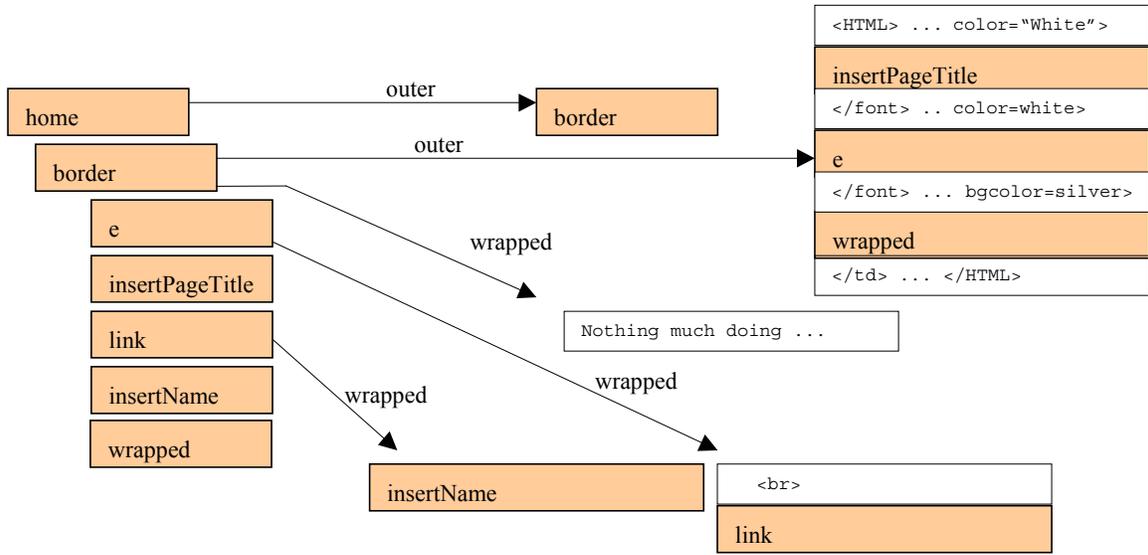
The left side shows how the components are embedded. The home component contains the border component. The border component contains the other components.

The outermost elements for the home component are a list of one item: the border component. There's no text before or after the border component's `<jwc>` tag, and the remaining text in the template is wrapped by the border component.

The border component is more complicated; it has several outer elements, representing static HTML text and components. The other components don't have templates, so they don't have outer elements, but some do wrap other components.

When it's time to render, the process starts in the home component. It iterates through its outer list, which contains a single element: border. We then recursively render the border component. Its outer list is rendered in order. Each component decides where it will render its wrapped elements (if it has any). The e component is a Foreach; it will render its wrapped elements any number of times.

Eventually, the process reaches the wrapped component, which is of type InsertWrapped. This is a special component; it jumps up one level to its container (the border) and renders the border's wrapped elements ... the text from the home page template.

It may seem complicated, but ultimately its very natural from a markup language point of view; the `<jwc>` tags continue to act like HTML elements, wrapping around and controlling their contents, just exactly like a `<table>` wraps its `<tr>`'s or a `<form>` wraps its `<input>`'s.

Localization

Tapestry has built in support for localization, designed to be easy to use. Tapestry allows multiple versions of HTML templates and assets (described in a later section) to be deployed with the application.

Each client connecting to the application will select a particular Locale. When a page for the application is created, the locale is used to select the correct template. Locales are defined by the ISO (International Standards Organization). A locale consists of a language code (such as 'en' for English, 'de' for German or 'fr' for French) and a country code (such as 'AU' for Australia, 'BE' for Belguim, or 'GB' for United Kingdom).

The base template name is derived from the specification name, by changing the '.jwc' extension to '.html'. For example, component `/com/skunkworx/skunkapp/Banner.jwc` will have a base template name of `/com/skunkworx/skunkapp/Banner.html`. This resource name is used as the basis of a search that includes the locale. Various suffixes are inserted just before the '.html' extension. A French speaking Belgian visitor would provoke the following search:

- `/com/skunkworx/skunkapp/Banner_fr_BE.html`

- `/com/skunkworx/skunkapp/Banner_fr.html`

- `/com/skunkworx/skunkapp/Banner.html`

The Tapestry framework automatically provides this basic level of support. To this, an application could add custom logic to allow the locale to be selected at runtime, and to store a user's locale preference (in a cookie, or in some form of profile) for later visits.

As of this writing, localization has not been implemented. However, support for localization is in all necessary interfaces.

Assets

Assets are images (GIF, JPEG, etc.), movies, sounds or other collateral associated with a web application. Assets come in three flavors: external, internal and private.

External assets live at an arbitrary URL. Internal assets use a URL within the servlet context hosting the Tapestry application; these assets are deployed on the same web server as the application, or within the same Web Application Archive (WAR).

Private assets come from the Java class path and are resources not normally visible to the web server.

Tapestry uses the assets concept to address two areas: localization and deployment.

For localization: internal and private assets are localized, just like HTML templates. That is, the path name provided is used as the basis for a search that takes into account the containing page's locale. External assets can't be localized in this way.

Private assets allow for easy deployment because the assets are packaged with the HTML templates and Java code of the application, inside a Java Archive (JAR) file, or within the `WEB-INF/classes` directory of a Web Application Archive (WAR) file.

The Tapestry framework provides two ways of exposing the assets to the client web browser.

First, it provides a service that will access the asset dynamically. The URL encodes the application servlet and the resource to download, and Tapestry framework code will pump the bytes down to the client web browser. This is the default behavior (and is most useful during development).

The second method involves copying the asset out to a directory visible to the web server, and creating a URL for it in its final location. This requires some extra configuration of the application. This method also has some implications when deploying new versions of the web application. These are addressed later in this document.

Component Specification

The component specification is an XML document. This discussion assumes a passing familiarity with XML documents. The specification is located inside the running Java VMs class path; in a deployed application, it will come out of the Java Archive (JAR) file or Web Application Archive (WAR).

The specification begins as follows:

```
<?xml version="1.0"?>
<!DOCTYPE specification PUBLIC
      "-//Primix Solutions//Tapestry Specification 1.0//EN"
      "http://tapestry.primix.com/dtd/Tapestry_1_0.dtd">

<specification>
```

This begins the XML document, and identifies it as a component specification.

Next, the type of the component, the Java class to instantiate, is defined.

```
<class>class name</class>
```

This must be a full class name, such as `com.skunkworx.skunkapp.Border`.

As previously described, the specification name is used to find the base HTML template name (by replacing the '.jwc' extension with '.html').

During development, such resources may be in the developer's work areas, but when the application is deployed, they will almost always be distributed inside a Java Archive (JAR) file or Web Application Archive (WAR) file. This is one of the ways that Tapestry eases deployment.

Next, the specification describes whether the component can have a body (that is, wrap around other HTML or components).

This is specified using the <allow-body> element:

```
<allow-body>boolean</allow-body>
```

The value specified should be "true" or "false". If not specified (the usual case), it defaults to true, allowing the component to have a body.

The next section describes the parameters to the component, all of which is specified inside a <parameters> element. Components can have two types of parameters: formal and informal. Formal parameters describe the interface to the component and have already been touched on.

Informal parameters are a kind of 'back door' to the HTML generated by the component. The majority of components have a one-to-one relationship to an HTML element.

Each informal parameter will be converted to an attribute / value pair and added to the HTML element. The component will silently filter out any informal parameters whose name conflicts with either a formal parameter name or an attribute controlled by the component. For example, the action component (which generates an <a> element) will not allow an informal parameter to specify the "href" attribute. Note that the check is case insensitive, so an informal parameter of "Href" would also be skipped.

Informal parameters are primarily used to set the "class" attribute (used in conjunction with cascading style sheets) and to create JavaScript event handlers.

Some components do not have the necessary one-to-one mapping to an HTML element that makes informal parameters useful. The <allow-informal-parameters> element controls this:

```
  <parameters>
      <allow-informal-parameters>boolean
      </allow-informal-parameters>
```

By default, informal parameters are allowed, so this is used to turn them off. This element is optional, and usually omitted.

Following this, each parameter is specified with a <parameter> element:

```
<parameter>
      <name>string</name>
      <java-type>class name</java-type>
      <required>boolean</required>
</parameter>
```

Each parameter must have a unique name. Parameter names must start with a letter and may contain letters, numbers and the underscore.

The Java type is the complete class name of a Java class that represents the type of data required (or provided) by the component. If not specified, it defaults to `java.lang.Object`.

If a parameter is required, then a binding must be provided for the parameter when it is embedded in another component. The `<required>` element is optional, and defaults to false (meaning the parameter is optional).

Once all formal parameters are defined, the parameters element is closed:

```
</parameters>
```

Following the parameters, embedded components are detailed. A `<components>` element surrounds a list of `<component>` elements.

A <component> tag starts each component:

```
<component>
  <id>component id</id>
  <type>component type</type>
```

Each component has a short id, which must be unique within its containing component. This id is used to match the component against a `<jwc>` tag in the HTML template.

Each component must also have a type. Types are resource paths for a component specification. Specifications end with '.jwc'. An example would be `/com/skunkworx/skunkapp/Banner.jwc`.

Unlike HTML templates, specifications are never localized.

As a convenience, aliases may be defined for components. An alias is a short name that takes the place of the specification resource path. All built-in components for Tapestry have aliases, and an application specification may define additional aliases.

For example, the component `/com/primix/tapestry/components/Action.jwc` has a standard alias of Action.

Most components have parameters, and therefore, bindings. Bindings are contained within a `<bindings>` element.

```
<bindings>
```

There are three different elements to specify bindings.

The `<binding>` element specifies a dynamic binding for the parameter.

```
<binding>
      <name>parameter name</name>
      <property-path>property path</property-path>
</binding>
```

The property path is specified relative to the containing component.

In some cases, the value for the parameter will never change; in this case a `<static-binding>` is used:

```
<static-binding>
      <name>parameter name</name>
      <value>value</value>
</static-binding>
```

Obviously, this only works for parameters that expect Strings or can coerce a String to a useful value (such as a boolean or int). Static bindings also work quite well with informal parameters.

Finally, in some cases, it makes sense for an embedded component to share a binding with its container. This is accomplished with the <inherited-binding> element:

```
<inherited-binding>
      <name>parameter name</name>
      <parameter-name>parameter name</parameter-name>
</inherited-binding>
```

In this case, the first parameter name is the name of a parameter of the embedded component. The second parameter name is the name of a parameter of the containing component.

Once all bindings for a component are specified, the `<bindings>` and `<component>` elements are closed.

```
  </bindings>
</component>
```

An example of this is a Border component, which has a title parameter that gives the name of the page. The Border component embeds an Insert component, and uses an inherited binding to set the Insert component's value parameter to the containing Border component's title parameter.

Additional components may be specified within their own `<component>` elements. Once all components are specified, the `<components>` element is closed.

```
</components>
```

Next, any assets for the component are specified. Assets are a way of identifying resources whose URLs will appear in a web page. Most often, the assets are image files used with an Image or Rollover component.

Assets may be stored at some arbitrary URL, may be somewhere on the same web site as the Tapestry application, or may be stored as a resource inside Java VM class path.

Assets are contained within an <assets> element.

```
<assets>
```

There are three types of elements for specifying assets:

```
<external-asset>
      <name>name</name>
      <URL>URL</URL>
```

```
</external-asset>

<internal-asset>
      <name>name</name>
      <path>path</path>
</internal-asset>

<private-asset>
      <name>name</name>
      <resource-path>resource path</resource-path>
</private-asset>
```

In all three cases, the name must be very simple: start with a letter and contain only letters, numbers and underscores. Assets names must be unique within the component.

For external assets, the URL must be complete: it will generally be inserted into the HTML unchanged.

For internal assets, the path must be relative to the web server's root directory. Internal assets may be localized; this will be reflected in the actual file chosen and in the URL inserted into the HTML.

For private assets, the resource path must be a resource path within the Java VM class path, like a specification or HTML template resource path. This means the image can be stored in the applications JAR, or in the `WEB-INF/classes` directory of the application's WAR, or inside some JAR in the class path.

Internal and private assets may be localized: Tapestry will perform a search similar to the one for HTML templates to locate the best match.

Once all assets have been specified, the <assets> element is closed:

```
</assets>
```

Since that's the last element in a component specification, the specification itself is closed:

```
</specification>
```

The major tags, `<parameters>`, `<components>`, `<bindings>` and `<assets>` are only specified if there are elements to be contained. A component with no components will not have a `<components>` element.   An embedded component with no bindings will not have a `<bindings>` element. In many cases, a component specification will be only a few lines long.

Chapter

4

Tapestry Pages

P ages are specialized versions of components. As components, they have a specification, embedded components, assets and an HTML template.

Pages do not have parameters, because they are the outermost component in the component hierarchy.

All components, however deep they're nesting, have a page property that points back to the page they are ultimately embedded within. Pages have an application property that points to the application they belong to.

Pages participate in a pooling mechanism, so that a single instance of a page component can be used by multiple sessions of the same web application. Even when a large number of client sessions are active, it is rare for more than a handful to be actively processing requests in the application server. This pooling mechanism minimizes the number of instances of a page that must exist concurrently on the server. There are some implications to this design that are discussed in the following sections.

Pages may have persistent state, properties that persist between request cycles. In fact, any component may have persistent state, and use the page as means for recording that state.

Application objects are session persistent objects, which is how they persist between request cycles. In some cases, an application object will be serialized in one Java VM and deserialized into another (for load balancing or fail-over reasons).

Pages are not session persistent. They exist only within the memory of the Java VM in which they are first created. Pages and components don't need to implement the `java.io.Serializable` interface; they will never be serialized.

Applications can always instantiate a new page instance and restore its previously recorded state (the recorded state information is serialized with the application).

Page Localization

When a page is first created, its locale is set to match the locale of the application it is loaded into. This page locale is read-only; it is set when the page is first created and never changes.

Any component or asset on the page that needs to be locale-specific (for instance, to load the correct HTML template) will reference the page's locale.

As noted previously, pages are not discarded; they are pooled for later reuse. When an application gets an existing page from the pool, it always matches its locale against the pooled page's locale. Thus a page and its application will always agree on locale, with one exception: if the application locale is changed during the request cycle.

When the application locale changes, any pages loaded in the current request cycle will reflect the prior locale. On subsequent request cycles, new pages will be loaded (or retrieved from the pool) with locales matching the application's new locale.

Tapestry does not currently have a mechanism for unloading a page in the same request cycle it was loaded (except at the end of the request cycle, when all pages are returned to the pool). If an application includes the ability to change locale, it should change to a new page after the locale change occurs.

Changing locale may have other, odd effects. If part of a page's persistent state is localized and the application locale is changed, then on a subsequent request cycle, the old localized state will be loaded into the new page (with the new locale). This may also affect any components on the page that have persistent state (though components with persistent state are quite rare).

In general, however, page localization is as easy as component localization and is usually not much of a consideration when designing web applications with Tapestry.

Persistent Page State

The Tapestry framework is responsible for tracking changes to page state during the request cycle, and storing that state between request cycles. This is accomplished through objects called page recorders. As a page's persistent state changes, it notifies the page recorder, providing the name of the property and the new value. Later, the page recorder can use this information to roll back the state of the page.

Pages are blind as to how their state is stored. The basic implementation of Tapestry simply stores the page state information in memory, but future options may include storing the data in flat files, relational databases or even as cookies in the client browser.

Persistence is provided by the application, which creates and manages the page recorders. In a simple application, the page recorders are serialized with the application object (for fail over and load balancing).

Some minor burden is placed on the page designer to support persistent state.

The mutator method of every persistent property must include a line of code that notifies the observer of the change.

For example, picture a page that has a persistent property for storing an email address. It would implement the normal accessor and mutator methods:

```
private String emailAddress;

public String getEmailAddress()
{
  return emailAddress;
}

public void setEmailAddress(String value)
{
  emailAddress = value;

  fireObservedChange("emailAddress", value);
}
```

The mutator method does slightly more than change the private instance variable; it must also notify the observer of the change, by invoking the method `fireObservedChange()`, which is implemented by the class `com.primix.tapestry.AbstractComponent`. This method is overloaded … implementations are provided for every type of scalar value, and for java.lang.Object.

The value itself must be serializable (scalar values are converted to wrapper classes, which are serializable).

The page designer must provide some additional code to manage the lifecycle of the page and its persistent properties. This is necessary to support the "shell game" that allows a page instance to be separate from its persistent state, and is best explained by example. Let's pretend that the user can select a personal preference for the color scheme of a page. The default color is blue.

The first user, Suzanne, reaches the page first. Disliking the blue color scheme, she uses a form on the page to select a green color scheme. The instance variable of the page is changed to green, and the page recorder inside Suzanne's session records that the persistent value for the color property is green.

When Suzanne revisits the page, an arbitrary instance of the page is taken from the pool. The page recorder changes the color of the page to green and Suzanne sees a green page.

However, if Nancy visits the same page for the first time, what is the color? Her page recorder will not note any particular selection for the page color property. She'll get whatever was left in the page's instance variable ... green if she gets the instance last used to display the page for Suzanne, or some other color if some other user recently hit the same page.

This may seem relatively minor when the persistent page state is just the background color. However, in a real application the persistent page state information may include user login information, credit card data, etc.

The way to deal with this properly is for each page with persistent state to override the method detachFromApplication(). The implementation should reset any instance variables on the page to their initial (freshly allocated) values.

In our example, when Suzanne is done with the page, its detachFromApplication() method will reset the page color property back to blue before releasing it into the pool. When Nancy hits the page for the first time, the page retrieved from the pool with have the expected blue property.

In our earlier email address example, the following code would be implemented by the page:

```
public void detachFromApplication()
{
  super.detachFromApplication();

  emailAddress = null;
}
```

Dynamic Page State

The properties of a page and components on the page can change during the rendering process. These are changes to the page's dynamic state.

The majority of components in an application use their bindings to pull data from the page (or from business objects reachable from the page).

A small number of components, notably the Foreach component, work the other way; pushing data back to the page (or some other component).

The Foreach component is used to loop over a set of items. It has one parameter from which it reads the list of items. A second parameter is used to write each item back to a property of its container.

For example, in our shopping cart example, we may use a Foreach to run through the list of line items in the shopping cart. Each line item identifies the product, cost and quantity.

```
Shopping cart for <jwc id="insertUserName"/>
<table>
  <tr> <th>Product</th> <th>Qty</th> <th>Price</th> </tr>
<jwc id="e-item">
<tr>
  <td> <jwc id="insertProductName"/> </td>
  <td> <jwc id="insertQuantity"/> </td>
  <td> <jwc id="insertPrice"/> </td>
  <td> <jwc id="removeAction">remove</jwc> </td>
</tr>
</jwc>
</table>
```

(Obviously, we've glossed over many details, such as allowing the quantity to be updated, and showing subtotals).

Component e-item is our Foreach.   It will render its body several times, depending on the number of line items in the cart. On each pass it:

- Gets the next value from the source

- Updates the value into some property of its container

- Renders its body (the static HTML and components it wraps)

This continues until there are no more values in its source. Lets say this is a page that has a lineItem property that is being updated by the e-item component. The insertProductName, insertQuantity and insertPrice components use dynamic bindings such as lineItem.productName, lineItem.quantity and lineItem.price.

Part of the page's specification would configure these embedded components:

```
<component>
  <id>e-item</id>
  <type>Foreach</type>

  <bindings>
    <binding>
      <name>source</name>
      <property-path>items</property-path>
    </binding>

    <binding>
      <name>value</name>
      <property-path>lineItem</property-path>
    </binding>
  </bindings>
</component>

<component>
  <id>insertProductName</id>
  <type>Insert</type>

  <bindings>
    <binding>
      <name>value</name>
      <property-path>lineItem.productName</property-path>
    </binding>
  </bindings>
</component>

<component>
  <id>insertQuantity</id>
  <type>Insert</type>

  <bindings>
    <binding>
      <name>value</name>
      <property-path>lineItem.quantity</property-path>
    </binding>
```

```
    </bindings>
</component>

<component>
  <id>insertPrice</id>
  <type>Insert</type>

  <bindings>
    <binding>
      <name>value</name>
      <property-path>lineItem.price</property-path>
    </binding>
  </bindings>
</component>

<component>
  <name>removeLineItem</name>
  <type>Action</type>

  <bindings>
    <binding>
      <name>listener</name>
      <property-path>removeListener</property-path>
    </binding>
  </bindings>
</component>
```

This is very important to the removeAction component. On some future request cycle, it will be expected to remove a specific line item from the shopping cart, but how will it know which one?

This is at the heart of the action service. One aspect of the `IRequestCycle's` functionality is to dole out a sequence of action ids that are used for this purpose (they are also involved in forms and form elements). As the action component renders itself, it allocates the next action id from the request cycle. Regardless of what path through the page's component hierarchy the rendering takes, the numbers are doled out in sequence. This includes conditional blocks and loops such as the Foreach.

The steps taken to render an HTML response are very deterministic. If it were possible to 'rewind the clock' and restore all the involved objects back to the same state (the same values for their instance variables) that they were just before the rendering took place, the end result would be the same. The exact same HTML response would be created.

This is similar to the way in which compiling a program for source code results in the same object code. Because the inputs are the same, the results will be identical.

This fact is exploited by the action service to respond to the URL. In fact, the state of the page and components is rolled back and the rendering processes fired again (with output discarded). Code inside the Action component can compare the action id against the target action id encoded within the URL. When a match is found, the Action component can count on the state of the page and all components on the page to be in the exact same state they were in when the page was previously rendered.

A small effort is required of the developer to always ensure that this rewind operation works. In cases where this can't be guaranteed (for instance, if the source of this dynamic data is a stock ticker or unpredictable database query) then other options must be used.

In our example, the removeAction component would trigger some application specific code implemented in its containing page that removes the current lineItem from the shopping cart.

This is usually implemented as an inner class implementing the IActionListener interface. Since our specification identified removeListener as the listener property for the removeLineItem component, the page must implement a remoteListener property:

```
public IActionListener getRemoveListener()
{
  return new IActionListener()
  {
    public void actionTriggered(IComponent component,
      IRequestCycle cycle)
      throws RequestCycleException
    {
      getCart().remove(lineItem);
    }
  };
}
```

This method is only invoked after all the page state is rewound; especially relevant is the lineItem property. The listener gets the shopping cart and removes from it the current line item. This whole rewinding process has ensured that lineItem is the correct value, even though the removeLineItem component was rendered several times on the page (because it was wrapped by the Foreach component).

An equivalent JavaServer Pages application would have needed to define a servlet for removing items from the cart, and would have had to encode in the URL some identifier for the item to be removed. The servlet would have to pick apart the URL to find the cart item identifier, locate the ShoppingCart (probably stored in the HttpSession) and the particular CartItem and invoke the remove() method directly. Finally, it would forward to the JSP that would produce the updated page.

The page containing the shopping cart would need to have special knowledge of the cart modifying servlet; its servlet prefix and the structure of the URL (that is, how the item to remove is identified). This creates a tight coupling between any page that wants to display the shopping cart and the servlet used to modify the shopping cart. If the shopping cart servlet is modified such that the URL it expects changes structure, all pages referencing the servlet will be broken.

Tapestry eliminates all of these issues, reducing the issue of manipulating the shopping cart down to the listener class, and its single, small method.

Page Versions and Stale Links

Each page recorder stores the version number of the page. This version number starts at 0 and is incremented in every request cycle when the persistent state of the page changes. Specifically, there's a point at which page changes are committed (just before the HTML response is rendered). It is at this point that the version number is incremented.

The action service relies on page state. The page state at the time the HTML response is created is encoded into the URL.

If an action service request URL references a page and the version number encoded in the URL does not match the actual page version number, then the application's stale-link page is used. This situation typically occurs when the user hits the back button on their web browser and selects a link from their history.

Page versions cannot account for any information that changes outside the Tapestry component model. For example, if part of the state of a page is based on a database query, and the results of that query change between request cycles (due to additions, deletions or modifications of rows in the result set), Tapestry will be unable detect that the actual page state has changed. This can lead to unexpected behavior in the application.

The direct service also encodes a page recorder version into its URLs. However, it doesn't reject URLs with out of date page versions; the information is noted and the request continues. Application logic can determine whether to reject the request or otherwise handle it. In most cases, direct actions don't care about either persistent or dynamic state, and the out of date version can be ignored.

Page Loading and Pooling

The process of loading a page (instantiating the page and its components) can be somewhat expensive. It involves reading the page's specification as well as the specification of all embedded components within the page. It also involves locating, reading and parsing the HTML templates of all components. Component bindings must be created and assigned.

All of this takes time ... not much time on an unloaded server but potentially longer than is acceptable on a busy site.

It would certainly be wasteful to create these pages just to discard them at the end of the request cycle.

Instead, pages are used during a request cycle, and then stored in a pool for later re-use. In practice, this means that a relatively small number of page objects can be shared, even when there are a large number of clients. The maximum number of instances of any one page is determined by the maximum number of clients that are processing a request that involves that page.

A page is taken out of the pool only long enough to process a request for a client that involves it. A page is involved in a request if it contains the component identified in the service URL, or if

application code involves the page explicitly (for instance, uses the page to render the HTML response). In either case, as so on as the response HTML stream is sent back to the client, any pages are released back to the pool.

This means that pages are out of the pool only for short periods of time. The duration of any single request should be very short, a matter of a second or two.

Page Buffering

The HTML response generated by a page during rendering is buffered. By default, eight kilobytes of 8-bit ASCII HTML is allowed to accumulate before any HTML output is actually sent back to the client web browser.

If a Java exception is thrown during the page rendering process, any buffered output is discarded, and the application-defined exception page is used to report the exception.

If a page generates a large amount of HTML and then throws an exception, the exception page is still used to report the exception, however the page finally viewed in the client browser will be "ugly", because part of the failed page's HTML will appear, then the complete HTML of the exception page.

Obviously, the exception page is a last case resort ... exceptions should be handled where they originate. Still, it is bad enough to present an exception page, it is worse to present a damaged exception page.

To avoid this scenario, the buffer size may be increased. If the buffer size is always larger than the amount of HTML generated by the page, then a failure anywhere on the page will still result in a clean exception page.

The buffer size can also be reduced, though in practice this has little effect. Page buffer size is set in the application specification.

There is a second aspect to page buffering; related to the HTTP protocol itself. An important optimization is the 'Keep Alive' session. When possible, one connection between a web browser and a web server should be used for multiple requests ... for example, to get a web page and then get all the images on the web page. Early web browsers and clients didn't support this behavior, so each request required opening a new connection to the server. All modern web browsers and servers will take advantage of Keep Alive sessions ... if possible.

For sessions to be kept alive, each request and response must identify the exact length of its content, in bytes, before transferring the content. For static web pages and static images, this is quite normal; the web browser simply looks up the size of the page or image before transferring its contents as a byte stream.

Dynamic web sites are different because, by their very nature, the number of bytes is not known when the first byte is written. For the server to indicate the end of the document to the client web browser, it must close the connection.

This is overcome by page buffering, where the first byte isn't written until the buffer is filled or the complete HTML response assembled in memory. If the entire HTML response is in the in-memory buffer before any of the response is returned to the client web browser, then the exact number of bytes will be known, just like a static page.

For critical pages such as the initial home page of an application, the page buffer should be set large enough to contain the entire page (this may require some experimentation to tune properly). Doing so should maximize the speed at which the initial page 'pops up' in the client web browser, since the existing connection will be reused by the browser to collect all the static assets ... the images and style sheets used on the page.

Chapter

5

Applications and Services

T he application object acts as a central hub for a single client's access to the Tapestry application.

Application objects are created by the application's servlet (described in the next section).

The application object provides support for the pages and components. It provides named *application services*, which are used to create and respond to URLs. It creates and manages the request cycle and provides robust default behavior for catching and reporting exceptions.

The application object provides the page recorder objects used by the request cycle. By doing so, it sets the persistence strategy for the application as a whole. For example, applications which subclass `com.primix.tapestry.app.SimpleApplication` will use the simple method of storing persistent state: in memory. Such applications may still be distributed, since the page recorders will be serialized with the application object (which is stored within the `HttpSession`).

Application Servlet

An application has a servlet, which acts as a bridge between the servlet container and the application object.

The servlet is responsible for locating the correct application object within the `HttpSession`. If necessary, it will create the application object and store it into the `HttpSession`.

Like most things in Tapestry, this is implied, not mandated, and can be avoided when necessary. For example, a very lightweight application that stores persistent page state in cookies may never need to create an `HttpSession` — but this is the exception to the rule. The general rule is that an `HttpSession` is created, the application located within it or created and stored into it.

The servlet may also perform some static initialization for the application in it's `init()` method. This includes tasks such as loading JDBC drivers.

In some cases, the application is dependent on some of the servlet's initialization parameters.

Servlet initial parameter name	Required	Description

com.primix.tapestry.asset.dir	No	The complete pathname of a directory to which private assets may be copied by the asset externalizer.
com.primix.tapestry.asset.URL	No	The URL corresponding to the external asset directory.
com.primix.tapestry.context-path	No	Needed only when deploying Tapestry as a Servlet 2.1 API web application. A servlet may only determine the context path (needed to generate URLs) under 2.2. Because Tapestry is compiled using 2.1, it may not use that method and this acts as a bridge.

Required Pages

Each application is required to have a minimum of four pages with specific names. Tapestry provides default implementations for three of the four, but a full-featured Tapestry application may override any of the other three to provide a consistent look-and-feel.

Tapestry only mandates the logical name of these four pages; the actual page component used is defined in the application specification.

The home page is the first page viewed by a client connecting to the application. Other than that, there is nothing special about the home page.

The initial connection to the application, where nothing is specified in the URL but the path to the servlet, causes the home service to be invoked, which makes use of the home page.

The restart service will also redirect the user to the home page.

No default is provided for the home page; every Tapestry application must define a home page.

The exception page is invoked whenever an exception is thrown when processing a service.

The Tapestry framework catches the exception and discards any HTML output (this is why output is buffered in memory).

The exception page must implement a writable JavaBeans property of type `java.lang.Throwable` named exception. The framework will invoke the accessor method before the page is rendered.

The class `com.primix.foundation.exception.ExceptionAnalyzer` and the component ExceptionDisplay are typically used to present this information.

The stale-link page is displayed when a `com.primix.tapestry.StaleLinkException` is thrown typically by an application service). The action service throws a StaleLinkException if the page version encoded in the URL doesn't match the actual page version.

The default implementation informs the user of the problem ("you really shouldn't use the back button on your browser") and uses the home service to send the client back to the home page.

The stale-session page is displayed when a `com.primix.tapestry.StaleSessionException` is thrown. This exception is thrown from the page, action and direct services if the `HttpSession` is newly created - this indicates a fresh connection to the servlet container after the old session timed out and was discarded.

Note that the home and restart services do not check for a newly created session.

Application Services

Application services provide the structure for building a web application from individual pages and components.

Each application service has a unique name. Well known names exist for the basic services (page, action, direct, etc., described in a later section).

Application services are responsible for creating URLs (which are inserted into the response HTML) and for later responding to those same URLs. This keeps the meaning of URLs localized. In a typical servlet or JSP application, code in one place creates the URL for some servlet to interpret. The servlet is in a completely different section of code. In situations where the servlet's behavior is extended, it may be necessary to change the structure of the URL the servlet processes ... and this requires finding every location such as URL is constructed and fixing it. This is the kind of inflexible, ad-hoc, buggy solution Tapestry is designed to eliminate.

Most services have a relationship to a particular component. The basic services (action, direct, page) each have a corresponding component (Action, Direct, Page). For example, to following example shows how the Page component is used to create a link between application pages.

First, an extract from the page's HTML template:

```
Click <jwc id="loginLink">here</jwc> to login.
```

Second, a portion of the page's specification:

```
<component>
  <id>loginLink</id>
  <type>Page</type>

  <bindings>
    <static-binding>
      <name>page</name>
      <value>login</value>
    </static-binding>
  </bindings>
</component>
```

The loginLink component will locate the page service, and provide 'login' (the name of the target

page) as a parameter. The page service will build and return an appropriate URL (*/servlet path*/page/login), which the loginLink component will incorporate into the `<a>` hyperlink it generates.

If the user later clicks the link, the application will invoke the page service to handle the URL; it will extract the page name ('login') and render that page.

The other services are more or less complicated, but share the same basic trait: the service provides the URL and later responds if the URL is triggered.

Application Specification

The application specification is used to identify the name of the application, the details of each page in the application, and to set up aliases for commonly used components within the application.

The application specification is loaded after the application object is created, which is very unlike pages and components (where the specification is read first and used to determine which class to instantiate).

As previously discussed, the application servlet is responsible for creating locating the application object or creating it.

The application specification begins with an XML header:

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC
        "-//Primix Solutions//Tapestry Specification 1.0//EN"
        "http://tapestry.primix.com/dtd/Tapestry_1_0.dtd">

<application>
```

The name of the application is specified. This name should be unique for all Tapestry applications running within a servlet container. It may contain letters, numbers, periods, spaces, underscores and dashes.

```
<name>application name</name>
```

Each page in the application is now defined, with a logical name:

```
<page>
  <name>page name</name>
  <specification-path>specification path</specification-path>
</page>
```

The name of the page must be unique within the application. It may contain letters, numbers, underscores, periods or dashes.

The specification path is the path of the component specification for the page, a resource within the Java class path. For example `/com/skunkworx/skunkapp/Home.jwc`.

Following this, aliases for components may be defined:

```
<component-alias>
  <alias>alias</alias>
  <specification-path>specification path</specification-path>
</component-alias>
```

This is used to create a short, memorable name for a frequently used component. Aliases have the same format as page names.

Component aliases are mostly freeform (letters, numbers and some punctuation are acceptable). A good guideline is to name the alias, the specification and the Java class the same. This naming pattern is used for all the internal Tapestry components. For example, the component `/com/primix/tapestry/components/Action.jwc` has an alias of Action and is implemented by the class `com.primix.tapestry.components.Action`.

Finally, the specification is completed by closing the application element:

```
</application>
```

Chapter

6

Understanding the Request Cycle

Web applications are significantly different in structure from other types of interactive applications. Because of the stateless nature of HTTP (the underlying communication protocol between web browsers and web servers), the server is constantly "picking up the pieces" of a conversation with the client.

This is complicated further in a high-end system that utilizes load balancing and fail over. In these cases, the server is expected to pick up a conversation started by some other server.

The Java Servlet API provides a base for managing the client - server interactions, by providing the `HttpSession` object, which is used to store server-side state information for a particular client.

Tapestry picks up from there, allowing the application, pages and components to believe (with just a little bit of work) that they are in continuous contact with the client web browser.

At the center of this is the request cycle. This request cycle is so fundamental under Tapestry that a particular object represents it, and it is used throughout the process of responding to a client request and creating an HTML response.

Each application service makes use of the request cycle in its own way. We'll describe the three common application services (page, action and direct), in detail.

In most cases, it is necessary to think in terms of two request cycles. In the first request cycle, a particular page is rendered and, along the way, any number of application service URLs are generated and included in the HTML. The second request cycle is triggered by one of those service URLs.

Page service

The page service is used for basic navigation between pages in the application. The page service is tightly tied to the Page component.

A page service URL is of the form:

```
/servlet path/page/page name
```

The request cycle for the page service is relatively simple.

- The service extracts the page name from the URL

- The named page it loaded and rolled back to its last recorded state

- The page is rendered to provide the HTML response sent back to the client

Direct service

The direct service is used to trigger a particular action. This service is tied to the Direct component.

A direct service URL is of the form:

```
/servlet path/direct/page name/page version/id path/context
```

The request cycle for the direct service is more complicated that the page service.

- The service extracts the page name from the URL

- The actual page version is checked again the URL; discrepancies are allowed but noted

- The page is loaded and its persistent state is rolled back to its last recorded state

- The service extracts the id path from the URL

- The component is located within the page, using its id path

- The component is notified that it has been triggered, with the context passed as an array of Strings

- The component locates its listener

- The listener is informed that the component was triggered

- The listener reacts in an application specific way, and may update the state of any number of pages

- The listener may optionally select a new page to render a response

- The state of any changed pages is recorded by the page recorders

- The result page (the page specified in the URL, unless overridden by the listener) is rendered to provide the HTML response sent back to the client

This is the way in which applications respond to the user. What's key is the component's listener, of type `com.primix.tapestry.components.IDirectListener`. This is the hook that allows pre-defined components from the Tapestry framework to access application specific behavior. The application or page objects (which are application specific) provide the necessary listener objects using dynamic bindings.

The direct service is useful in many cases, but does have its limitations. The state of the page when the listener is invoked is its state just prior to rendering (in the previous request cycle). This can cause a problem when the action to be performed is reliant on state that changes during the rendering of the page. In those cases, the action service (and Action component) should be used.

The Direct component has an optional parameter named 'parameters'. The value for this may be a single String, an array of Strings, or a Collection of Strings. These strings are added to the URL after the id path. When the action is triggered, the array is reconstructed (from the URL) and provided to the Direct component and its listener. This is primarily used to encode dynamic page state directly into the URL when doing so is not compatible with the action service (described in the next section).

Action service

The action service is also used to trigger a particular application-specific action using an Action component, and its listener. The action service is also used by the Form component to process HTML form submissions.

An action service URL is of the form:

```
/servlet prefix/action/page name/page version/action id
```

The request cycle for the action service is more complicated that the direct service. This sequence assumes that the component is an Action, the details of handling form submissions are described in a later section.

- The service extracts the page name from the URL

- The actual page version is checked again the URL[1]

---

[1] This is used to detect "stale links", cases where the user has navigated backwards using their browser, back in time, to a view of the page with a prior state. Unfortunately, the page version may not be enough ... in some cases the dynamic state of the page is based on objects that are not in changed properties of the page, such as in the application object or in an EJB. This is an area of Tapestry requiring more work.

- The page is loaded and rolled back to its last recorded state

- The service extracts the action id from the URL

- A render of the page is started, with output discarded

- Each component "goes through the motions" of rendering

- Eventually, the Action component identifies that the page has been rewound (the action id in the URL matches the current action id)

- The Action component locates its listener

- The listener is informed that the action was triggered

- The listener reacts in an application specific way, and may update the state of any number of pages

- The listener may optionally select a new page to render a response

- The Action component terminates the rewind cycle

- The state of any changed pages is recorded by the page recorders

- The result page (the page specified in the URL, unless overridden by the listener) is rendered to provide the HTML response sent back to the client

The process of restoring the page's dynamic state is called rewinding. Rewinding is used to go beyond restoring a page's persistent state and actually restore the page's dynamic state. Whatever state the page was in when the action URL was rendered in the previous request cycle is restored before the action's listener is invoked.

Use of the action service is convenient, but not always appropriate. Deeply nested Foreach components will result in a geometric increase in processing time to respond to actions (as well as render the HTML).

If the data on the page is not easily accessible then the action service should be avoided. For example, if the page is generated from a long running database query. Alternate measures, such as storing the results of the query as persistent page state should be used.

Another alternative is to use the direct service (and Direct component) instead, as it allows the necessary context to be encoded into the URL. This can be very useful when the dynamic state of the page is dependant on expensive or unpredictably changing data (such as a database query).

For example, a product catalog could encode the primary key of the products listed as direct service parameters to create links to a product details page.

Action service and forms

Processing of requests for Form components is a little more complicated than for ordinary Action components. This is because a Form will wrap a number of form-related components, such as TextField, Select, Option, Checkbox and others.

In order to accept the results posted in the HTML form, each of these components must be given a chance respond to the request. A component responds to the request by extracting a request parameter from the `HttpServletRequest`, interpreting it, and assigning a value through a parameter. Most Form related components also have an optional listener, which will be notified after the value is assigned.

As with an Action component again, a full rewind must be done, to account for conditional portions of the page and Foreaches.

The Form component doesn't terminate the rewind cycle until after all of its wrapped components have had a chance to render. It then notifies its own listener.

The basic components, Text and TextField, are quite simple. They are simply moving text between the application, the HTML and the submitted request.

Individual Checkbox components are also simple: they set a boolean property.

The Option and Radio components work the same way; they set a boolean property. This makes perfect sense from the point of view of the individual component, but may not always be what the application developer needs.

For a Radio component, or an Option component when multiple selection is not allowed, the end user will select one possible value from a list. The resulting behavior on the application side is to turn off all boolean properties but one.

An application-specific subclass of `com.primix.tapestry.components.SelectionAdaptor` can be used in combination with the Radio or Option to translate between application objects and this virtual array of flags.

More often, a form will use a group of radio buttons or a pop-up option list, to select a single value to assign to some object property; the PropertySelection component is designed to efficiently handle this situation.

Provided Components

T
he Tapestry framework provides a large set of pre-written components to use as a starting point for building applications, or more complicated components.

This section will provide a brief overview of the available components and their most commonly used parameters. The Javadoc is the final reference for this information. All components live in the `com.primix.tapestry.components` package.

Each of these components is automatically added as an alias in an application specification.

Action

Creates an `<a>` hyperlink in the HTML response. If the link is triggered, then the Action retrieves its listener, and invokes `actionTriggered()` on it.

Parameter	Description
listener	The object notified when the link is triggered.
enabled	Controls whether the link is produced, if false, then the link is not created, but the contents are still rendered.

Any

Used to make an existing HTML element dynamic. Informal parameters are used to fill in the attributes of the element.

Parameter	Description
element	The element to be produced

Body

Replaces the <body> tag. The Body component provides some support for other components, such as the Rollover component.

Checkbox

Used to implement a checkbox form element. Must be wrapped by a Form component.

Parameter	Description
selected	Indicates whether the checkbox is selected or not.
disabled	Controls whether the text field is active or not.

Conditional

Makes part of an HTML template conditional. The body of the Conditional component is only rendered if a specified condition is met.

Parameter	Description
condition	The condition to be met.
invert	If true, then the condition is inverted. This is useful for simulating and else clause.

DatabaseQuery

A wrapper around a JDBC query. It converts each row selected from the database into a Map that it provides to its container, much like a Foreach component.

Parameter	Description
connection	The JDBC Connection to execute the query on.
query	A SQL query to execute.
row	A property to update with each row from the query.

Delegator

A component which delegates its behavior (that is, the HTML renders) to some other object. This is occasionally useful when the HTML to render is best generated from pure Java code (or has been provided from some source outside of Tapestry).

Parameter	Description
delegate	The object which will provide the rendering for the component.

Direct

Creates a <a> hyperlink that notifies the component when the link is triggered. The link includes some context-specific data that is passed on to the component's listener. This is used in cases where the Action component can't be used (or is to inefficient).

Parameter	Description
listener	Specifies an object that is notified when the link is clicked.
context	An array of Strings to be encoded into the URL. These parameters will be decoded when the link is triggered.
enabled	Controls whether the link is produced. If disabled, the portion of the template the link surrounds is still rendered, but not the link itself.

ExceptionDisplay

A component, used on the standard exception page, used to render a detailed list of the exceptions.

Parameter	Description
exceptions	An array of `com.primix.foundation.exception.ExceptionDescription` objects

Foreach

A component that loops through a set of values, setting a property for each value before rendering its wrapped elements.

Parameter	Description
source	The source of objects to be iterated.
value	Used to update the current value on each iteration.

Form

A component that manages an HTML form. The other form-related components must be wrapped inside the Form.

Parameter	Description
method	The value to use for the method attribute of the `<form>` tag.
listener	The listener, informed **after** the wrapped components of the form have had a chance to absorb the request.

Image

Used to insert a static image.

Parameter	Description
image	The image to show, specified as an asset.

ImageButton

An image button inside a Form. Clicking the button submits the form.

Parameter	Description
image	The image to show, specified as an asset.
listener	A listener that can respond when this component is used to submit the form.

Insert

Inserts dynamic text into the HTML response. Invalid characters (such as < and >) are converted into HTML entities (`<` and `>`).

Parameter	Description
value	The value to be inserted, which may be a string or an arbitrary object (that is converted to a string).

InsertURL

Used to insert some the URL from an asset

Parameter	Description
asset	The asset whose URL is to be inserted.

InsertWrapped

Renders the static HTML and components wrapped by the container of the InsertWrapped component. Components which have HTML templates and can wrap other elements should include exactly one InsertWrapped component in their template.

Option

Implements a component that manages an HTML <option> form element, and must be wrapped by a Select component.

Parameter	Description
selected	Used to indicate whether the given option is selected.

Page

Creates a link to another page within the application.

Parameter	Description
page	The name of a page to link to.
enabled	Controls whether the link is produced. If disabled, the portion of the template the link surrounds is still rendered, but not the link itself.

PropertySelection

Creates either a <select> or a sequence of <input type=radio> controls that ultimately allow a property to be set. Uses a model to map between Java values that will be assigned, and textual labels that will appear in the HTML response.

Parameter	Description
value	The property value to edit.
model	The model used to enumerated the potential values, and to produce text labels for each value.

radio	If true, the component is rendered as a sequence of radio buttons. If false (the default), the component is rendered as a <select>

Radio

Implements a component that manages an HTML form element. Such a component must be wrapped (possibly indirectly) inside a RadioGroup component.

Parameter	Description
selected	Used to indicate whether the given option is selected. Only updated is the Radio is not disabled.
disabled	If true, then the Radio is disabled. It will write a disabled attribute into its tag when rendering, and will not update its selected binding.

RadioGroup

A container of Radio components, which itself must be within a Form component.

Rollover

Works with any kind of link component, which must wrap it, to provide a rollover image button … a button where the image changes when the mouse move over it or off of it. A Rollover is disabled if the link that wraps it is disabled.

Parameter	Description
image	The (initial) image to show (if not disabled).
focus	The image displayed when the mouse moves over the button (unless disabled).
blur	The image displayed when the mouse moves off of the button (unless disabled).
disabled	The image displayed if the link is itself disabled.

Select

A wrapper around Option components, within a Form.

Parameter	Description

multiple	If true, the component allows multiple selection.

Service

A component for creating a link for an arbitrary service that uses no parameters, such as the reset or home service.

Parameter	Description
service	The name of the service.
enabled	Controls whether the link is produced. If disabled, the portion of the template the link surrounds is still rendered, but not the link itself.

Text

Implements a component that manages an HTML <textarea> form element, within a Form component.

Parameter	Description
text	The text inside the textarea.
disabled	Controls whether the textarea is active or not.
columns	The width of the textarea, in characters.
rows	The number of rows in the textarea.

TextField

Implements a component that manages an HTML `<input type=text>` or `<input type=password>` form element., with a Form component.

Parameter	Description
text	The text inside the textarea.
hidden	If true, then the text field is written as a `<input type=password>` form element.
disabled	Controls whether the text field is active or not.

50

displayWidth	Controls the display width of the text control in the client browser.
maximumWidth	Controls the maximum characters that the text control will accept

Chapter

8

Designing Tapestry Applications

When first starting to design a Tapestry application, the designer consider some basic elements as a guide to the overall design of the application.

Persistent Storage Strategy

Tapestry pages store a certain amount of client state between request cycles. The Tapestry application chooses the strategy for this storage by choosing a particular super-class.

Currently, only the `com.primix.tapestry.app.SimpleApplication` class is provided with the framework; it uses in-memory page recorders. When the application object is serialized, the page recorders are serialized along with it.

The `IPageRecorder` interface doesn't specify anything about how a page recorder works. This opens up many possibilities for storage of state, including flat files, databases, stateful EJB session beans, or HTTP Cookies.

In fact, a very sophisticated application could mix and match, using cookies for some pages, in-memory for others.

By default, page recorders stay active for the duration of the user session. If a page will not be referenced again, or its persistent state is no longer relevant or needed, the application may explicitly "forget" its state.

Remember that for load balancing and fail over reasons, the application object will be serialized and de-serialized. Ideally, its serialized state should be less than two kilobytes; because Java serialization is inefficient, this does not leave much room.

Identify Pages and Page Flow

Early in the design process, the page flow of the application should be identified. Each page should be identified and given a specific name.

Page names are less structured than other identifiers in Tapestry. They may contain letters, numbers, underscores, dashes and periods. Tapestry makes absolutely no interpretation on the page names.

In many applications, certain parts of the functionality are implemented as "wizards", several related pages that are used in sequence as part of a business process. A common example of this is initial user registration, or when submitting an order to an e-commerce system.

A good page naming convention for this case is "*wizard-name.page-name*" (a period separates the two names). This visually identifies that several pages are related. In addition, a Java package for the wizard should be created to contain the Java classes, component specifications, HTML templates and other assets related to the wizard.

The designer must also account for additional entry points to the Application beyond the standard home page. These will require additional application services (see below).

Identify Common Logic

Many applications will have common logic that appears on many pages. For example, an e-commerce system may have a shopping cart, and have many different places where an item can be added to the cart.

In many cases, the logic for this can be centralized in the application object. The application could implement methods for adding products to the shopping cart. This could take the form of Java methods, component listeners, or even new application services.

In addition, most web applications have a concept of a 'user'. The object representing the user should be a property of the application, making it accessible to all pages and components.

Identify Application Services

Tapestry applications will need to define new application services when a page must be referenced from outside the Tapestry application

This is best explained by example. It is reasonable in an e-commerce system that there is a particular page that shows product information for a particular product. This information includes description, price, availability, user reviews, etc. A user may want to bookmark that page and return to it on a later session.

Tapestry doesn't normally allow this; the page may be book marked, but when the bookmark is triggered, the user will be told that they have a "stale session". The URLs normally generated in a

Tapestry application are very context sensitive; they are only meaningful in terms of the user's navigation throughout the application, starting with the home page. When book marked, that context is lost.

The built-in services (page, action and direct), all reject any URL if there isn't an existing HttpSession. They redirect to a stale session page, that warns the use of what has occurred and allows them to start the application fresh (a standard page for this is provided by Tapestry, but can be overridden by an application that wants to provide different content or look-and-feel).

By defining a new application service, the necessary context can be encoded directly into the URL, and is similar to how the direct action works. This is partially a step backwards towards typical servlet or JSP development, but even here Tapestry offers superior services. In the e-commerce example, the application service URL could encode some form of product identifier.

Identify Common Components

Even before detailed design of an application, certain portions of pages will be common to most, if not all, pages. The canonical example is a "navigation bar", a collection of links and buttons used to navigate to specific pages within the application. An e-commerce site may have a shopping cart related component that can appear in many places.

In many cases, common components may need to be parameterized: the navigation bar may need a parameter to specify what pages are to appear; the shopping cart component will require a shopping cart object (the component is the view and controller, the shopping cart object is the model).

Chapter

9

Coding Tapestry Applications

Application Object

Application objects are somewhat different than page and component objects. The latter are created after their specification is read, using information inside the specification.

Application objects are created by the application servlet. Only after they are created do they locate or parse their specification.

Most of the time, the specification does not have to be read, it is located instead. The first time the specification is read, it is stored into the `ServletContext` under a well-known name. When a subsequent instance of the same application is created, it can find the specification in the ServletContext rather than have to parse it.

The application itself specifies the well-known name, by implementing the method `getSpecificationAttributeName()`. A single `ServletContext` may simultaneously run multiple Tapestry applications, so the value returned from this method should be unique (incorporating the class name or package is recommended).

However, if not already stored in the ServletContext, the `getSpecificationResourceName()` method is invoked to get the resource path of the specification.

Application objects will be serialized and de-serialized as part of load balancing and fail over. As much as possible, attributes of the application object should be transient. For example, the instance variable that holds the `ApplicationSpecification` is transient; if needed (after de-serialization), the application can locate the specification in the `ServletContext` or re-parse it from the specification file.

Chapter

10

Designing new components

Creating new components using Tapestry is designed to be quite simple.

Components are typically created through aggregation, that is, by combining existing components using an HTML template and specification.

You will almost always want to define a short alias for your new component in the application specification. This insulates developers from minor name changes to the component specification, such as moving the component to a different Java package.

Like pages, components should reset their state back to default values when the page they are contained within is returned to the pool.

Most components do not have any meaningful state. A component which does should implement the `ILifecycle` interface, and implement the `reset()` method.

The `reset()` method is invoked from the page's `detachFromApplication()` method, which is invoked at the very end of the request cycle, just before the page is returned to the page pool.

Choosing a base class

There are two basic types of components: those that use an HTML template, and those that don't.

Nearly all of the base components provided with the Tapestry framework don't use templates. They inherit from the class `com.primix.tapestry.AbstractComponent`. Such components must implement the `render()` method.

Components that use templates inherit from a subclass of AbstractComponent: `com.primix.tapestry.BaseComponent`. They should leave the `render()` method alone (it already does what it should ... load the template if necessary and render itself using the template).

In some cases, a new component can be written just by combining existing components (this often involves using inherited bindings). Such a codeless component will consist of just a specification and an HTML template and will use the BaseComponent class.

Parameters and Bindings

You may create a component that has parameters. Under Tapestry, component parameters are a kind of "named slot" that can be wired up as a source (or sink) of data in a number of ways. This "wiring up" is actually accomplished using binding objects.

The page loader, the object that converts a component specification into an actual component, is responsible for creating and assigning the bindings. It uses the method `setBinding()` to assign a binding with a name. Your component can retrieve the binding using `getBinding()`.

For example, lets create a component that allows the color of a span of text to be specified using a `java.awt.Color` object. The component has a required parameter named 'color'. The class's render() method is below:

```
public void render(IResponseWriter writer, IRequestCycle cycle)
  throws RequestCycleException
{
  IBinding colorBinding;
  Color color;
  String encodedColor;

  colorBinding = getBinding("color");
  color = (Color)colorBinding.getValue();
  encodedColor = RequestContext.encodeColor(color);

  writer.begin("font");
  writer.attribute("color", encodedColor);

  renderWrapped(writer, cycle);

  writer.end();
}
```

The call to `getBinding()` is relatively expensive, since it involves rummaging around in a `java.util.Map` and then casting the result from `java.lang.Object` to `com.primix.tapestry.IBinding`.

Because bindings are typically set once and the read frequently by the component, implementing them as private instance variables is much more efficient. Tapestry allows for this as an optimization on frequently used components.

The `setBinding()` method in `AbstractComponent` checks to see if there is a read/write JavaBeans property named "*name*Binding" for type `com.primix.tapestry.IBinding`. In this example, it would look for the methods `getColorBinding()` and `setColorBinding()`.

If the methods are found, they are invoked from `getBinding()` and `setBinding()` instead of updating the `Map`.

 This changes the example to:

```
private IBinding colorBinding;
```

```
public void setColorBinding(IBinding value)
{
  colorBinding = value;
}

public IBinding getColorBinding()
{
  return colorBinding;
}

public void render(IResponseWriter writer, IRequestCycle cycle)
  throws RequestCycleException
{
  Color color;
  String encodedColor;

  color = (Color)colorBinding.getValue();
  encodedColor = RequestContext.encodeColor(color);

  writer.begin("font");
  writer.attribute("color", encodedColor);

  renderWrapped(writer, cycle);

  writer.end();
}
```

This is a trade off; slightly more code for slightly better performance. There is also a memory bonus; the Map used by AbstractComponent will never be created.

Persistent Component State

Like pages (described further in the document), individual components may have state that persists between request cycles. This is rare for non-page components, but still possible and useful in special circumstances.

A client that must persist some client state uses its page's changeObserver. It simply posts ObservedChangeEvents with itself (not its page) as the source. In practice, it still simply invokes the fireObservedChange() method.

Component Assets

Tapestry components are designed for easy re-use. Most components consist of a specification, a Java class and an HTML template.

Some components may need more; they may have additional image files, sounds, Flash animations, QuickTime movies or whatever. These are collectively called "assets".

Assets come in three flavors: external, internal and private.

- An external asset is just a fancy way of packaging a URL at an arbitrary web site.

- An internal asset represents a file with a URL relative to the web server containing the Tapestry application.

- A private asset is a file within the class path, that is, packaged with the component in a Java Archive (JAR) file. Obviously, such assets are not normally visible to the web server.

Components which use assets don't care what flavor they are; they simply rely on the method `buildURL()` to provide a URL they can incorporate into the HTML they generate. For example, the Image component has an image parameter that is used to build the src attribute of an HTML `<img>` element.

Assets figure prominently into three areas: reuse, deployment and localization.

Internal and private assets may be localized: when needed, a search occurs for a localized version, relative to a base name provided in the component specification.

Private assets simplify both re-use and deployment. They allow a re-usable Tapestry component, even one with associated images, style sheets (or other assets) to be incorporated into a Tapestry application without any special consideration.

In a traditional web application, private assets would need to be packaged separately from the 'component' code and placed into some pre-defined directory visible to the web server.

Under Tapestry, the private assets are distributed with the component specification, HTML templates and Java code, within a Java Archive (JAR) file, or within the WEB-INF/classes directory of a Web Application Archive (WAR) file. The resources are located within the running application's class path.

The Tapestry framework takes care of making the private assets visible to the client web browser. This occurs in one of two ways:

- The private assets are copied out of the class path and to a directory visible to the web server. This requires some additional configuration of the Tapestry application servlet.

- The assets are dynamically accessed from the class path using the asset service.

Copying assets out of the class path and onto the web site is the best solution for final deployment, since it allows the assets to be retrieved as static files, an operation most web servers are optimized for.

Dynamically accessing assets requires additional operations in Java code. These operations are not nearly as efficient as static access. However, dynamic access is much more convenient during development since much less configuration (in this case, copying of assets) is necessary before testing the application.

As with many things in Tapestry, the components using assets are blind as to how the assets are made visible to the client.

Finally, every component has an assets property that is an unmodifiable `java.util.Map`. The assets in the `Map` are accessible as if they were properties of the `Map`. In other words, the property path assets.welcome is valid, if the component defines an asset named 'welcome'.