# Tapestry Developer's Guide

**Howard Lewis Ship**

# Tapestry Developer's Guide

Howard Lewis Ship

# Table of Contents


1. Introduction ..... 1
    - Scripting vs. Components ..... 1
    - Interaction ..... 4
    - Security ..... 5
    - Web Applications ..... 5
    - Features ..... 6
2. JavaBeans and Properties ..... 8
    - JavaBeans ..... 8
    - JavaBeans Properties ..... 8
    - Property Paths ..... 10
    - Object Graph Navigation Library ..... 10
3. Tapestry Components ..... 11
    - Parameters and Bindings ..... 11
    - Connected Parameters ..... 12
    - Formal vs. Informal Parameters ..... 14
    - Embedded Components ..... 14
    - HTML Templates ..... 15
        - Localizing sections of a template ..... 16
        - Components with Bodies ..... 16
    - Tapestry and HTML Production ..... 17
        - Implicitly removed bodies ..... 17
        - Explicitly removed bodies ..... 18
        - Limiting template content ..... 18
        - Limits ..... 19
    - Localization ..... 19
        - Localization with Strings ..... 19
        - Localization with Templates ..... 20
    - Assets ..... 20
    - Helper Beans ..... 21
4. Tapestry Pages ..... 22
    - Page State ..... 22
    - Persistent Page State ..... 23
    - EJB Page Properties ..... 24
    - Dynamic Page State ..... 25
    - Stale Links and the Browser Back Button ..... 27
    - Page Loading and Pooling ..... 28
    - Page Localization ..... 30
    - Page Buffering ..... 31
    - Page Events ..... 31
5. Application Engines and Services ..... 34
    - Application Servlet ..... 34
    - Required Pages ..... 35
    - Server-Side State ..... 37
    - Stateful vs. Stateless ..... 37
    - Engine Services ..... 37
    - Logging ..... 39
    - Private Assets ..... 40
6. Understanding the Request Cycle ..... 41
    - Service URLs and query parameters ..... 41
    - Page service ..... 41
    - Action and Direct listeners ..... 42
    - Direct service ..... 44
    - Action service ..... 45

Services and forms ..... 47
7. Designing Tapestry Applications ..... 48
Persistent Storage Strategy ..... 48
Identify Pages and Page Flow ..... 48
Identify Common Logic ..... 48
Identify Engine Services ..... 49
Identify Common Components ..... 50
8. Coding Tapestry Applications ..... 51
Application Engine ..... 51
Visit Object ..... 51
Operating Stateless ..... 51
Enterprise JavaBeans Support ..... 52
Page classes ..... 52
9. Designing new components ..... 53
Choosing a base class ..... 53
Parameters and Bindings ..... 53
Persistent Component State ..... 55
Component Assets ..... 55
10. Tapestry and JavaScript ..... 57
The Body component ..... 58
Script Specifications and Script Components ..... 59
11. The Tapestry Inspector ..... 60
Specification View ..... 60
Template View ..... 60
Properties View ..... 61
Engine View ..... 62
Logging View ..... 63
A. Tapestry JAR files ..... 65
B. Tapestry Specification DTDs ..... 66
<application> element ..... 66
<bean> element ..... 67
<binding> element ..... 68
<configure> element ..... 68
<component> element ..... 69
<component-alias> element ..... 70
<component-specification> element ..... 70
<context-asset> element ..... 71
<description> element ..... 72
<extension> element ..... 72
<external-asset> element ..... 73
<field-binding> element ..... 74
<inherited-binding> element ..... 74
<library> element ..... 75
<library-specification> element ..... 75
<page> element ..... 75
<page-specification> element ..... 76
<parameter> element ..... 76
<private-asset> element ..... 78
<property> element ..... 78
<reserved-parameter> element ..... 79
<service> element ..... 79
<set-property> element ..... 80
<set-string-property> element ..... 80
<static-binding> element ..... 80
<string-binding> element ..... 81
C. Tapestry Script Specification DTD ..... 82
<body> element ..... 82
<foreach> element ..... 82

`<if>` element ..... 83
`<if-not>` element ..... 83
`<include-script>` element ..... 84
`<initialization>` element ..... 84
`<input-symbol>` element ..... 84
`<let>` element ..... 85
`<script>` element ..... 85
`<set>` element ..... 86

# List of Figures


4.1. Page Lifecycle ..... 29
4.2. Page Loading Sequence ..... 29
4.3. Page Render Sequence ..... 31
4.4. Page Rewind Sequence ..... 32
4.5. Page Detach Sequence ..... 33
5.1. ApplicationServlet Sequence ..... 35
5.2. Services and Gestures ..... 38
6.1. Page Service Sequence ..... 42
6.2. Direct Service Sequence ..... 44
6.3. Action Service Sequence ..... 45
10.1. Body Component Rendering Sequence ..... 58
11.1. Inspector - Specification View ..... 60
11.2. Inspector - Template View ..... 60
11.3. Inspector - Properties View ..... 61
11.4. Inspector - Engine View ..... 62
11.5. Inspector - Logging View ..... 63
B.1. `<application>` Attributes ..... 66
B.2. `<application>` Elements ..... 67
B.3. `<bean>` Attributes ..... 68
B.4. `<bean>` Elements ..... 68
B.5. `<binding>` Attributes ..... 68
B.6. `<configure>` Attributes ..... 69
B.7. `<component>` Attributes ..... 69
B.8. `<component>` Elements ..... 70
B.9. `<component-alias>` Attributes ..... 70
B.10. `<component-specification>` Attributes ..... 70
B.11. `<component-specification>` Elements ..... 71
B.12. `<context-asset>` Attributes ..... 72
B.13. `<description>` element ..... 72
B.14. `<extension>` Attributes ..... 73
B.15. `<component-specification>` Elements ..... 73
B.16. `<external-asset>` Attributes ..... 73
B.17. `<field-binding>` Attributes ..... 74
B.18. `<inherited-binding>` Attributes ..... 74
B.19. `<library>` Attributes ..... 75
B.20. `<library-specification>` Elements ..... 75
B.21. `<page>` Attributes ..... 76
B.22. `<page-specification>` Attributes ..... 76
B.23. `<page-specification>` Elements ..... 76
B.24. `<parameter>` Attributes ..... 77
B.25. `<private-asset>` Attributes ..... 78
B.26. `<property>` Attributes ..... 78
B.27. `<reserved-parameter>` Attributes ..... 79
B.28. `<service>` Attributes ..... 79
B.29. `<set-property>` Attributes ..... 80
B.30. `<set-string-property>` Attributes ..... 80
B.31. `<static-binding>` Attributes ..... 80
B.32. `<string-binding>` Attributes ..... 81
C.1. `<body>` Elements ..... 82
C.2. `<foreach>` Attributes ..... 82
C.3. `<foreach>` Elements ..... 83
C.4. `<if>` Attributes ..... 83
C.5. `<if>` Elements ..... 83

C.6. <if-not> Attributes ..... 83
C.7. <if-not> Elements ..... 83
C.8. <include-script> Attributes ..... 84
C.9. <initialization> Elements ..... 84
C.10. <input-symbol> Attributes ..... 84
C.11. <let> Attributes ..... 85
C.12. <let> Elements ..... 85
C.13. <script> Elements ..... 85
C.14. <set> Attributes ..... 86

# List of Tables

5.1. Tapestry Pages ..... 35
B.1. Tapestry Specifications ..... 66

# List of Examples

2.1. JavaBeans getter method ..... 8
2.2. JavaBeans setter method ..... 8
2.3. JavaBeans getter method (boolean) ..... 9
2.4. Lazy evaulation of JavaBeans property ..... 9
2.5. Synthesized JavaBeans Property ..... 9
3.1. Connected Parameter - Specification ..... 13
3.2. Connected Parameter - Java Code ..... 13
4.1. HTML template for Shopping Cart ..... 25
4.2. Shopping Cart Specification (excerpt) ..... 26
4.3. Listener method for remove component ..... 27
5.1. Web Deployment Descriptor ..... 34
10.1. Traditional JavaScript usage ..... 57

# Chapter 1. Introduction

Tapestry 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 2.2 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.

Tapestry reconceptualizes web application development in terms of objects, methods and properties instead of URLs and query parameters.

# 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
- Allaire ColdFusion
- PHP
- WebMacro
- FreeMarker
- Velocity

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 *emphasised* 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 works quite differently; it allows existing tags to be marked as dynamic in a completely unobtrusive way.

A Tapestry component is any HTML tag with a `jwcid` attribute ("jwc" stands for "Java Web Component"). For comparison, an equivalent Tapestry template to the previous JSP example:

```
<h1>Contents of shopping cart for
<span jwcid="insertUserName">John Doe</span>:</h1>
```

This defines a component named `insertUserName` on the page. To assist HTML development, a sample value, "`John Doe`" is included, but this is automatically editted out when the HTML template is used by the framework.

The `<span>` tag simply indicated where the Tapestry component will go ... it doesn't identify any of its behavior. That is provided elsewhere, in a component specification.

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

```
<component id="insertUserName" type="Insert"> ❶
  <binding name="value" expression="visit.userName"/>  ❷
</component>
```

❶ The `id` attribute gives the component a unique identifier, that matches against the HTML template. The `type` attribute is used to specify which kind of component is to be used.

❷ Bindings identify how the component gets the data it needs. In this example, the `Insert` component requires a binding for its `value` parameter, which is what will be inserted into the response HTML page. This type of binding (there are others), extracts the userName property from the visit object (a central, application-defined object used to store most server-side state in a Tapestry application).

Tapestry really excels when it is doing something more complicated than simply producing output. For example, let's assume that there's a checkout button that should only be enabled when the user has items in their shopping cart.

In the JSP world, this would look something like:

```
<%
   boolean showLink;
   String imageURL;
   showLink = applicationObject.getHasCheckoutItems();
   if (showLink)
     imageURL = "/images/Checkout.gif";
   else
     imageURL = "/images/Checkout-disabled.gif";

  if (showLink)
  {
     String linkURL;
     linkURL = response.encodeURL("/servlet/checkout"); %>
<a href="<%= linkURL %>">
<% } %>
<img border=0 src="<%= imageURL %>" alt="Checkout"><%
  if (showLink)
    out.println("</a>");
%>
```

This assumes that `applicationObject` exists to determine whether the user has entered any checkout items. Presumably, this object was provided by a controlling servlet, or placed into the `HttpSession`.

The corresponding Tapestry HTML template is much simpler:

```
<a jwcid="checkoutLink"><img jwcid="checkoutButton" alt="Checkout"/></a>
```

A bit more goes into the page's specification :

```
<component id="checkoutLink" type="PageLink"> ❶
  <static-binding name="page">Checkout</static-binding>
  <binding name="disabled" expression="visit.cartEmpty"/> ❷
</component>

<component id="checkoutButton" type="Rollover"> ❸
  <binding name="image" expression="assets.checkout"/>
  <binding name="disabled" expression="assets.checkoutDisabled"/>
</component>

<external-asset name="checkout" URL="/images/Checkout.gif"/> ❹
<external-asset name="checkoutDisabled" URL="/images/Checkout-disabled.gif"/>
```

❶ Component `checkoutLink` is a `PageLink`, a component that creates a link to another page in the application. Tapestry takes care of generating the appropriate URL.

❷ The `disabled` parameter allows the link to be "turned off"; here it is turned off when the shopping cart is empty.

❸ A `Rollover` component inserts an image; it must be inside some kind of link component (such as the `PageLink`) and is sensitive to whether the link is enabled or disabled; inserting a different image when disabled. Not shown here is the ability of the `Rollover` component to generate dynamic mouse-over effects as well.

❹ Tapestry uses an abstraction, assets, to identify images, stylesheets and other resources. The `Rollover` component wants a reference to an asset, not a URL.

The point of this example is that the JSP developer had to worry about character-by-character production of HTML. Further, the ratio of Java code to HTML is quickly getting out of hand.

By contrast, the Tapestry developer is concerned with the behavior of components and has an elegant way of specifying that behavior dynamically.

# 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> <span jwcid="insertName">Sample Product</span> </td>
    <td> <span jwcid="insertQuantity">10</span> </td>
    <td> <a jwcid="remove">Remove</a> </td>
</tr>
```

Because of the component object model used by Tapestry, the framework knows exactly "where on the page" the `remove` component is. It uses this information to build an appropriate URL that references the `remove` component. If the user clicks the link, the framework will inform the component to perform the desired action. The `remove` 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 and deploy. 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.

Tapestry applications are built on top of the Java Servlet API, and so inherits all the sercurity benefits of servlets. Most security intrusions against CGI programs (such as those written in Perl or other scripting languages) 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.

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. Again, this is a common limitation of servlet applications in general.

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.

# Web Applications

Tapestry has a very strong sense of what an application is, derived from an XML specification file. This file identifies and gives names to all the pages in the application, and identifies certain other key classes as well. It also gives a human-readable name to the entire application.

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 a shopping cart (in a typical e-commerce application). 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. These constraints are intended to be of minimal impact to the developer, imposing an acceptible amount of structure. They create a common language that can be used between members of a team, and even between the technical and creative groups within a 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 implementing similar constructs.

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.

At the core of any Tapestry application are two objects: the engine and the visit. The engine is created when the first request from a client arrives at the server. The engine is responsible for all the mundane tasks in Tapestry, such as managing the request cycle. It is sort of a dispatcher, that handles the incoming request and runs the process of responding to the request with a new HTML page.

The visit is a second object that contains application-specific data and logic. Its type is completely defined by the application. In an e-commerce application, the visit might store a shopping cart and information about the user (once logged in).

Both the engine and the visit are stored persistently between request cycles, inside the `HttpSession` object.

The engine 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.

# 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 [1]
- Easy deployment
- The Inspector, which allows developers to debug a running Tapestry application

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
- XML / XHTML support
- Improved WAP / WML support
- A real-time performance "Dashboard"

[1] That is, Tapestry templates and specifications are interpreted as is. Unlike JSPs, they are not translated into Java source code and compiled into Java classes. This greatly simplifies debugging since no code is generated at runtime.

# Chapter 2. JavaBeans and Properties

The Tapestry framework is based upon the use of JavaBeans and JavaBeans properties. This chapter is a short review of these concepts. A more involved discussion is available as part of the Java Tutorial.

# JavaBeans

The JavaBeans framework is a way of manipulating Java objects when their exact type is not known. The ability to make objects work together, when their exact type is not known, is very powerful. It's an example of the kind of flexibility availble in a highly dynamic language like Java that is not possible in lower-level languages such as C++.

The JavaBeans framework is the basis for a number of component-based frameworks, including Java's AWT and Swing GUI libraries, as well as Tapestry. The idea is that, by following a few naming rules and coding conventions, it is possible to "plug into" a framework with new classes, classes not even written yet when the framework is created. In Tapestry terms, this is used to allow the creation of new Tapestry components.

Any Java object can act as a JavaBean; it just has to follow certain naming conventions (discussed in the next section). In cases where a framework needs to create new instances of a class, such as when Tapestry creates a new instance of a component, the Java class must implement a public, no arguments constructor (it may implement additional constructors as well).

The Java Reflection API allows Tapestry to access the methods, attributes and constructors of a class.

# JavaBeans Properties

For Tapestry, the central concept for JavaBeans are properties. The JavaBeans framework allows Tapestry to treat any object as a collection of named properties. Tapestry frequently reads, and occasionally writes, values from or to these named properties.

A property is *not* the same as an *attribute* ... though, most often, each property is backed up by an attribute. To Tapestry, and the Reflection API, a property is a set of public methods on the object. Accessing a property involves invoking one of these methods.

### Example 2.1. JavaBeans getter method

```
public type getName()
{
  ...
}
```

### Example 2.2. JavaBeans setter method

```
public void setName(type value)
{
  ...
}
```

A property may be read-only or write-only (that is, it may implement just one of the two methods). The *`type`* may be a scalar type (boolean, int, etc.) or any Java class.

Note the naming; the first letter of the property name is capitalized after `get` or `set`. JavaBeans properties are case sensitive with respect to the method names and the property names. A special case exists when the name is an acronyn; this is recognized by two or more upper-case letters in a row (after get or set); in this case, the property name does *not* have the first letter convert to lower-case.

As a special case, a boolean property may use an alternate name for the getter method:

### Example 2.3. JavaBeans getter method (boolean)

```
public boolean is<i>Name</i>()
{
  ...
}
```

Although the normal implementation is to get or set an instance variable, more complicated options are possible. One pattern is *lazy evaluation*, where an expensive calculation is put off until the actual value is needed, for example:

### Example 2.4. Lazy evaulation of JavaBeans property

```
public List userNames = null;

/**
 *  Returns a List of user names obtained from the database.
 *
 **/

public List getUserNames()
{
  if (userNames == null)
    userNames = fetchUserNamesFromDatabase();

  return userNames;
}
```

Here, the first time the method is invoked, the expensive database fetch occurs. The value returned from the database is then cached for later invocations.

Another common pattern is a *synthesized property*. Here, there is no real attribute at all, the value is always computed on the fly. A frequent use of this is to avoid tripping over null pointers.

### Example 2.5. Synthesized JavaBeans Property

```
/**
 *  Returns the name of the company's account representative, if
 *  if the company has one, or null otherwise.
 *
 **/

public String getAccountRepName()
{
  AccountRep rep = company.getAccountRep();

  if (rep == null)
    return null;

  return rep.getName();
}
```

This example creates a synthetic property, `accountRepName`.

# Property Paths

The JavaBeans framework provides basic named properties for JavaBean objects. Tapestry extends this from simple properties to *property paths*.

A property path is a series of property names, separated by periods. When reading a property path, each property is read in series.

In the example from the introduction, the property path `visit.userName` was referenced. This path means that the `visit` property of the start object (a Tapestry page) should be accessed, then the `userName` property of the visit object should be accessed. This is approximately the same as Java code `getVisit().getUserName()` (except that property access is not typesafe).

In some cases, property paths are used to change a value, instead of reading it. When this occurs, all properties but the last a read; only the last property is written. In other words, updating `visit.userName` would be similar to the JavaCode `getVisit().setUserName(`*`value`*`)`.

Property paths can be of any length; however, they are just as susceptable to `NullPointerException`s as any other JavaCode. Care must be taken that none of the properties in a property path, except the final one, return null. This can often be accomplished using synthesized properties.

# Object Graph Navigation Library

Beyond even simple property paths are the powerful Object Graph Navigation Library (OGNL) *expressions*. OGNL expressions are modeled after Java expressions; they can invoke methods, perform comparisons, do arithmetic ... even build collections on the fly.

OGNL is a separate framework from Tapestry; further details about it are available at http://www.ognl.org.

# Chapter 3. 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 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 into a component and doing something with it.

For example, on a welcome page, a component might get the `userName` property from the `visit` object and insert it into the HTML response.

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 completely 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. The binding is a bridge between the component and the parameter value, exposing that value to the component as it is needed. The reason for all this is to allow pages, and the components within them, to be shared by many concurrent sessions ... a major facet in Tapestry's strategy for maintaining application scalability.

When a component needs the value of one of its parameters, it must obtain the correct binding, an instance of interface `IBinding`, and invoke methods on the binding to get the value from the binding. Additional methods are used with output parameters to update the binding property.

In most cases, discussed in the next section, Tapestry can hide the bindings from the developer. In effect, it automates the process of obtaining the binding, obtaining the value from it, and assigning it to a JavaBean property of the component.

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. The source of this data is a property of some component.

In fact, dynamic bindings use property paths, allowing a binding to 'crawl' deeply through an object graph to access the value it needs. This frees the components from relying totally on the properties of their container, instead they are free to access properties of more distant objects.

# Connected Parameters

In most cases, a developer is not interested in bindings; an easier model for developers is one in which Tapestry uses the parameters and bindings to set properties of the component automatically. Starting in release 2.1, Tapestry includes this behavior, with some constraints and limitations.

Part of the `<parameter>` specification for a parameter is the *direction*, which can be one of the following values:

in
Input parameter; the value is drawn from the binding (if bound) and applied to the corresponding component property just before rendering the component.

form
A parameter which matches the semantics of a form component. The parameter is treated like an `in` parameter when the page is rendering.

When the form containing the component is submitted, the connected property is read (after the component renders), and the value applied to the parameter.

custom
Tapestry does not try to connect the parameter with any property; the component is responsible for accessing the binding and retrieving or setting values.

This type must be used for any kind of output parameter, or for an input parameter where the property may be accessed other than during the rendering of the component.

## Why aren't output parameters connectable?

The problem is the timing of output parameters. Sometimes a parameter is only an output parameter when the containing form is submitted (for example, any of the form related components). Sometimes a parameter is output many times (for example, `Foreach`) while the component renders.

The latter case may always be handled as custom; the former case may be handled in the future.

Defining a parameter as direction `in` causes Tapestry to connect the parameter to the corresponding property of the component. The parameter specification must identify the Java type of the property. Properties must be read/write (they must have both getter and setter methods).

Tapestry will set properties from parameters just before rendering the component. After the component renders, the parameters are cleared; they are returned to inital values. Tapestry reads these initial values just before it sets the properties the first time. This makes it very easy to set defaults for optional parameters: just provide a default value for the correspoinding instance variable.

If the property is connected to an invariant binding (a static or field binding), then the property is set just once, and never cleared.

There are times when the parameter name can't be used as the property name. For example, the `PageLink` component has a `page` parameter, the name of the page to link to. However, all components already have a `page` property, the `IPage` that ultimately contains them. The specification for the `PageLink` component connects the `page` parameter to a property named `targetPage` instead.

Defining a connected parameter as required means that the parameter must be bound *and* the binding

must provide a non-null value. A runtime exception is thrown when a required parameter's binding yields a null value.

The following examples show how to declare and use a parameter:

### Example 3.1. Connected Parameter - Specification

```
<specification ...>
  <parameter name="color" direction="in" java-type="java.awt.Color"/>
  ...
```

### Example 3.2. Connected Parameter - Java Code

```
public class ColorComponent extends AbstractComponent
{
  private Color color = Color.RED;

  public Color getColor()
  {
    return color;
  }

  public void setColor(Color color)
  {
    this.color = color;
  }

  protected void renderComponent(IMarkupWriter writer, IRequestCycle cycle)
  throws RequestCycleException
  {
    writer.begin("font");
    writer.attribute("color", ^RequestContext;.encodeColor(color);

    renderWrapped(writer, cycle);

    writer.end();
  }

}
```

In this example, the component writes its content inside a `<font>` element, with the HTML color attribute set from the `color` parameter. `RequestContext` includes a static convienience method for converting from a `Color` object to an encoded color that will be meaningful to a web browser.

The parameter is optional and defaults to red if not specified (that is, if the parameter is not bound).

At runtime, Tapestry will invoke `setColor()` first (if the `color` parameter is bound). It will then in-

voke `renderComponent()`. Finally (even if `renderComponent()` throws an exception) it will invoke `setColor()` again, to restore it back to the default value, `Color.RED`.

This code includes a defect: because the parameter is optional, there is nothing to prevent it from being bound to null.

# Formal vs. Informal Parameters

Tapestry components have two types of parameters: formal and informal.

Formal parameters are parameters defined in the component specification. Each formal parameter has a specific (case sensitive) name and may be required or optional.

In many cases, there is a one-to-one mapping between a Tapestry component and a specific HTML tag. For example, `Body` and `<body>`, `Form` and `<form>`, etc. In other cases, a Tapestry component produces a known single HTML tag. For example, `ActionLink`, `DirectLink`, `PageLink` and `ServiceLink` all produce an `<a>` tag.

To support truly rich interfaces, it is often necessary to specify additional attributes of the HTML tags; usually this means setting the `class` of a tag so as to get visual properties from a stylesheet. In other cases, display attributes may be specified inline (this is often the case with attributes related to display width and height, since CSS support for these properties are inconsistent between the major HTML 4.0 browsers).

In theory, these components *could* define additional formal parameters for each possible HTML attribute ... but there are a huge number of possible attributes, many of which are specific to a particular browser.

Instead, Tapestry has the concept of an *informal parameter*. This is an "additional" parameter, not specified in the component's specification. In most cases, where informal parameters are allowed, they are added as additional HTML attributes (there are a few special exceptions, such as the `Script` component).

Informal parameters do have some limitations. Informal parameters that conflict with the names of any formal parameters, or with any of the HTML attributes generated directly by the component, are silently ommitted. The comparison is case-insensitve. Thus, for a `DirectLink` component, you can not change the `href` attribute, even if you supply a `Href` (or other variation) informal parameter.

Not all Tapestry components even allow informal parameters; this is explicitly stated in the component specification.

### Informal Parameters that are Assets

Tapestry includes a special case when an informal parameter is actually an asset. The URL for the asset is determined and that is the value supplied for the attribute.

# Embedded Components

Under Tapestry, it is common to define new components by combining existing components. The existing components are embedded in the containing component. This is always true at the top level; Pages, which are still Tapestry components, always embed other Tapestry components.

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, but their internals, how they are implemented, are not revealed.

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 constructing 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 [1] 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.

One of the features of Tapestry is *invisible instrumentation*. In most web application frameworks, converting a static HTML page into a usable template is a destructive process: the addition of new tags, directives or even Java code to the template means that it will no longer preview properly in a WYSIWYG editor.

Tapestry templates are instrumented using a new HTML attribute, `jwcid`, to any existing element. Elements with such attributes are recognized by Tapestry as being dynamic, and driven by a Tapestry component, but a WYSIWYG editor will simply ignore them. Once a template is instrumented, it may be worked on by both the HTML producer and the Java developer.

Identifying a Tapestry component within an HTML template is accomplished by adding a `jwcid` attribute to a tag.

```
<any jwcid="component id" ... >  body  </any>
```

or

```
<any jwcid="component id" ... />
```

Most often, the HTML element chosen is `<span>`, though (in fact) Tapestry completely ignores the element chosen by the developer, except to make sure the open and close tags balance.

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

You are free to specify additional attributes. These attributes will become informal parameters for the

[1] The current releases of Tapestry is specifically oriented around HTML. Some support for non-HTML languages, such as XML, XHTML or WML is already present and will be expanded in the future.

Tapestry component.

The start and end tags for Tapestry components must balance properly. This includes cases where the end tag is normally ommitted, such as `<input>` elements. Either a closing tag must be supplied, or the XML-style syntax for an empty element must be used (that is, a slash just before the end of the tag).

# Localizing sections of a template

Tapestry includes an additional template feature to assist with localization of a web application. By specifying a `<span>` element with a special attribute, `key`, Tapestry will replace the entire `<span>` tag with a localized string for the component.

This construct takes one of two forms:

```
<span key="key" ... > ... </span>
```

or

```
<span key="key" ... />
```

If only the `key` attribute is specified, then the `<span>` is simply replaced with the localized string. However, if any additional attributes are specified for the `<span>` tag beyond `key`, then the `<span>` tag will be part of the rendered HTML, with the specified attributes.

The upshot of this is that sections of the HTML template can be invisibly localized simply by wrapping the text to be replaced inside a `<span>` tag. The wrapped text exists, once more, as sample text to be displayed in a WYSIWYG editor.

# Components with Bodies

In Tapestry, individual components may have their own HTML templates. This is a very powerful concept ... it allows powerful and useful components to be created with very little code. By contrast, accomplishing the same using JSP tags requires either that all the HTML be output from the JSP tag directly, or that the JSP tag use some additional framework, such as Velocity, to enable the use of a template. In either case the JSP tag author will need to divide the code or template into two pieces (before the body and after the body). Tapestry allows components to simply have a single template, with a marker for where the body is placed.

During the rendering of a page, Tapestry knits together the templates of the page and all the nested components to create the HTML response sent back to the client web browser.

```
Container content ❶
<span jwcid="component"> ❷
  Body content ❸
</span>
More container content ❹
```

❶ This portion of the container content is rendered first.
❷ The component is then rendered. It will render, possibly using its own template.
❸ The component controls *if*, *when* and *how often* the body content from its container is rendered.

Body content can be a mix of static HTML and additional components. These components are *wrapped* by the component, but are *embedded* in the component's container.
❹ After the component finishes rendering, the remaining content from the container is rendered.

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 allows a body (and wrap other elements), or whether it discards it, is defined in the component's specification.

Tapestry includes a special component, `RenderBody`, which is used to render the body content from a component's container. It makes it easy to create components that wrap other components.

# Tapestry and HTML Production

Tapestry is design to work in a large-scale environment, that typically features two seperate teams: a "creative" team that produces HTML and a "technical" team that produces Tapestry pages, components and Java code.

The division of skills is such that the creative team has virtually no knowledge of Java and a minimal understanding of Tapestry, and the technical team has a limited understanding of HTML (and tend to be color blind).

The typical workflow is that the technical team implements the application, using very minimal HTML ... that is, minimal attention to layout, font size, colors, etc. Just enough to be sure that the functionality of the application is there.

Meanwhile, the creative team is producing HTML pages of what the finished application will look like. These pages are like snapshots of the HTML produced by the running application.

*Integration* is the process of merging these two views of the application together. Primarily, this involves marking up tags within the HTML page with `jwcid` attributes, to indicate to Tapestry which portions of the page are dynamic. In this way, the page can be used as a Tapestry HTML template. These changes are designed to be invisible to a WYSIWYG HTML editor.

Tapestry includes a number of additional features to allow the HTML producers to continue working on HTML templates, *even after* their initial efforts have been integrated with the Java developer's code.

## Implicitly removed bodies

In many cases, a component doesn't allow a body, but one may be present in the HTML template. As usual, this is declared in the component's specification. Tapestry considers that body to be a sample value, one which exists to allow the HTML producer to verify the layout of the page using a WYSIWYG editor (rather than having to run the entire application). Tapestry simply edits out the body at runtime.

For example, an HTML producer may create an HTML template that includes a table cell to display the user's name. The producer includes a sample value so that the cell isn't empty (when previewing the HTML layout).

```
<td><span jwcid="insertName">John Doe</span></td>
```

The `Insert` component doesn't allow a body, so Tapestry edits out the content of the `<span>` tag from the HTML template. The fact that a `<span>` was used to represent the `Insert` component in the HTML template is irrelevant to Tapestry; any tag could have been used, Tapestry just cares that the start and end tags balance.

At runtime, Tapestry will combine the HTML template and the `Insert` component to produce the final HTML:

```
<td>Frank N. Furter</td>
```

This editting out isn't limited to simple text; any HTML inside the body is removed. However, none of that content may be dynamic ... the presence of a `jwcid` attribute will cause a parsing exception.

# Explicitly removed bodies

Another feature related to production and integration is the ability to remove sections of the HTML template. Producers often include some optional portions on the page. The canonical example of this is a page that shows a table of results; the HTML producer will usually include extra rows to demonstrate the look and layout of a fully populated page.

The first row will be wrapped by a `Foreach` and otherwise changed to include dynamic links and output, but what about the other rows?

To handle this case, Tapestry recognizes a special `jwcid` attribute value: `$remove$`. Using this special id causes Tapestry to edit out the tag and all of its contents. Thus, each additional `<tr>` in the table should specify the value `$remove$` for attribute `jwcid`.

```
<table>
  <tr jwcid="foreach">
    <td><span jwcid="insertUserName">John Doe</span></td>
    <td><span jwcid="insertAge">42</span></td>
  </tr>
  <tr jwcid="$remove$">
          <td>Frank N. Furter</td>
          <td>47</td>
  </tr>
  <tr jwcid="$remove$">
    <td>Bob Doyle</td>
    <td>24</td>
  </tr>
</table>
```

# Limiting template content

In a typical Tapestry application, some form of Border component provides a significant portion of every page. This typically includes the outermost `<html>`, `<head>` and `<body>` tags, as well as

<table>s used to control layout.

In the static HTML pages from the creative team, this is not directly visible ... they *must* include all the content normally generated by the Border component in order to see what the HTML page actually looks like.

By default, the *entire* HTML template is the content for the page. This causes a problem, even after a  is added, to represent the Border component ... much of the HTML is duplicated, once from the static HTML, then dynamically from the Border component.

To eliminate this problem, Tapestry has a second special `jwcid` attribute: `$content$`. Using this special id causes Tapestry to limit its view of the HTML template to just the content inside the tag. Anything outside the defined content is completely ignored.

## Limits

Ideally, the HTML pages created by the HTML producers would be used as is as the HTML templates. Changes made for integration, the adding of `jwcid` attributes and such, would be copied back into the HTML pages.

Given the use of the `$remove$` and `$content$` `jwcid`'s, this is practical to a point. Once the application starts using a number of re-usable components, there isn't a good way to perform the integration short of cutting and replacing some of the HTML page content to form the HTML template.

# Localization

Tapestry has built in support for localization, designed to be easy to use. This localization support is defined in terms of transforming the user interface into a format appropriate the the locale of the user. This primarily takes the form of localized text (translated into the end-user's language), but can also affect other aspects of look and feel including colors, images and layout.

Tapestry has two different methods for supporting localization; developers are free to mix and match solutions according to their own preferences.

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 localized resources. 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).

A client's initial locale is determined by analyzing HTTP headers provided with the initial request. An application may override this default, which records a client-side cookie identifyng the desired locale. An example of this is included in the Tapestry Workbench demonstration.

## Localization with Strings

Each individual component may have a set of *localized strings*. Remember that pages are just a specific kind of component. This set is built, much like the properties of a `ResourceBundle`, from one or more `.properties` files. These files are located on the classpath, in the same directory as the component specification (the `.jwc` file).

The search for strings is much the same as with `ResourceBundle`, except that only `.properties` files are considered (`ResourceBundle` also looks for classes).

Example: for a component `/com/skunkworx/skunkapp/Border.jwc` and a locale of `fr_BE` would be:

- `/com/skunkworx/skunkapp/Border_fr_BE.properties`
- `/com/skunkworx/skunkapp/Border_fr.properties`
- `/com/skunkworx/skunkapp/Border.properties`

Searching for individual keys works just as with `ResourceBundle`, the search starts in the most specific file (`Border_fr_BE.properties`) and continues downward if not found.

Components can gain access to their container's localized strings via the `<string-binding>` element in the component specification.

# Localization with Templates

Tapestry allows multiple versions of HTML templates and assets (described in a later section) to be deployed with the application.

The base template name is derived from the specification name, by changing the `jwc` extension to `html`. For example, component `/com/skunkworx/skunkapp/Border.jwc` will have a base template name of `/com/skunkworx/skunkapp/Border.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/Border_fr_BE.html`
- `/com/skunkworx/skunkapp/Border_fr.html`
- `/com/skunkworx/skunkapp/Border.html`

### Note

This form of localization actually predates the alternate form, using localized strings. Localizing the strings seperately from the rest of the HTML template is generally a better and easier way. Localization of templates will, in the future, be used primarily when changing the layout of the template ... for example, to provide a right-to-left orientation in a Hebrew localization.

# Assets

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

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

Private assets come from the Java classpath 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 desired 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.

Private assets support re-usability; a re-usable component may be packaged with supporting assets (typically, image files) and used in any Tapestry application without change, and without having to locate, extract or otherwise fiddle with those assets.

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.

# Helper Beans

There is a second form of aggregation allowed with Tapestry components. The first way, covered previously, is to use embedded components to extend the functionality of the outer component. In some cases, useful behavior can be isolated, not into an additional component, but into a simple JavaBean.

These additional beans, called helper beans, are defined in the component specification, in the `<bean>` element. Each bean has a unique name, a class to instantiate, and a lifecycle (which controls how long the component keeps a reference to the bean). The specification allows properties of the bean to be set as well, using the `<set-property>` and `<set-string-property>` elements. Helper beans are accessed through the `beans` property of the component.

Beans are created as needed, they may then be cached for future use according to their declared lifecycle. The default lifecycle is `request`, meaning that the same bean will be returned until the end of the current request cycle.

An alternate lifecycle, `page`, means that once the bean is instantiated, it will continue to be available for the lifetime of the page containing it. Remember that helper beans should never contain any client-specific state, since a page will be used by multiple sessions and clients.

The last available lifecycle, `none`, indicates that the bean is not cached at all, and will be created fresh on each property access.

Tapestry includes a handful of useful helper beans. `Default` is used to provide default values for optional parameters. `ValidationDelegate` and several implementions of `IValidator` used with `ValidField`, it allows simple handling of validation and presentating validation errors. `EvenOdd` is used by the Tapestry Inspector; it generates a stream of values alternating between "even" and "odd"; this is combined with cascading stylesheets to make the rows alternate between white and grey backgrounds.

# Chapter 4. Tapestry Pages

Pages 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 their nesting, have a page property that points back to the page they are ultimately embedded within. Pages have an engine property that points to the engine they are currently attached 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 specific to a particular user that persist between request cycles. These properties live only as long as the `HttpSession`. There is some complexity here, because the page state is entirely *seperate* from any instance of the page. Remember that on subsequent requests, a different page from the page pool may be used to service the request ... in fact, in a clustering environment, the request may be serviced by an entirely different server. Tapestry efficiently and transparently hides these details; when any portion of an application requests a page, it receives an instance of the page with all persistent page properties set the the values previously stored for the user.

In fact, any component may have persistent state, and use the page as means for recording that state.

The engine is a session persistent object. The implementation of this varies from application server to application server, but the basic idea is that the `HttpSession` is serialized after each request and stored in a file or database. It may then be removed from memory. When a subsequent request for the same session arrives, it is restored from the persistent storage.

In a clustering server application, consequtive requests for the same session may be serviced by different servers within the cluster. Serializing and deserializing the `HttpSession` is the mechanism by which the servers are kept synchronized. Persistent page properties are stored as part of the engine, and so they continue to be available, even after the engine has moved from one server to another.

The visit object is a property of the engine object, so it is serialized and de-serialized with the engine.

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.

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

# Page State

Pages, and the components on them, have state. State is considered the set of values for the properties of the page.

In Tapestry, the lifespan of each property is very important. There are three lifespans:

- Persistent. Changes the property are recorded and persist between request cycles. Persistent properties are restored when the page is next loaded. Persistent properties are specific to an individual user.

- Transient. The property is set before the page is rendered and will be reset (to its default value) at the end of the current request cycle.
- Dynamic. The property changes even while the page is rendered, but (like transient) the property is reset at the end of the current request cycle.

Persistent properties are things like the user's name, the product being displayed in an e-commerce application, etc. Transient properties are more commonly things needed just once, such as an error message. Dynamic properties are intimately tied to the rendering process ... for example, to display a list of items in an order, it may be necessary to have a dynamic property take the value of each line item in sequence, as part of a loop.

# 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. Ultimately, this is the responsiblility of the application engine. This is accomplished through page recorder objects. As a page's persistent state changes, it notifies its page recorder, providing the name of the property and the new value.

This information is stored persistently between request cycles. In a later request cycle, the page recorder combines this information with a page instance to rollback 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 (and serializes it with the engine, in the `HttpSession`), but future options may include storing the data in flat files, relational databases or even as cookies in the client browser.

Some minor burden is placed on the developer 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, consider 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 `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, the contents of a shopping cart or whatever. The way to deal with this properly is for each page with persistent state to override the method `detach()`. 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 `detach()` 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 other words, it is the responsibility of the developer to ensure that, as a page is returned to the pool, its state is exactly the same as a freshly created page.

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

```
public void detach()
{
  emailAddress = null;

  super.detach();
}
```

All properties, dynamic, transient and persistent, should be reset inside the `detach()` method.

Individual components on a page may also have dynamic, transient or persistent properties. If so, they should implement the `PageDetachListener` interface and implement the `pageDetached()` method and clear out such properties, just as a page does in `detach()`.

# EJB Page Properties

Tapestry make a single, special case for one particular type of persistent page property: references to Enterprise JavaBeans.

The page recorders check to see if a page property is type `javax.ejb.EJBObject`. If so, they don't store the object itself (`EJBObjects` are not directly serializable), instead they get the `Handle` for the object and store that instead (`Handle`s are serializable).

When the page is next accessed, the `Handle` is converted back into an `EJBObject` before assigning it

to the page property.

A side effect of this is that you may not have a `Handle` as a persistant page property; the page recorders don't have a way to differentiate a `Handle` from an `EJBObject` converted to a `Handle` and always assume the latter.

# 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.

### Example 4.1. HTML template for Shopping Cart

```
<h1>Context of shopping cart for
<span jwcid="insertUserName">John Doe</span></h1>
<table>
  <tr>
    <th>Product</th> <th>Qty</th> <th>Price</th>
  </tr>
  <span jwcid="eachItem">
  <tr>
    <td><span jwcid="insertProductName">Product Name</span></td>
    <td><span jwcid="insertQuantity">5</span></td>
    <td><span jwcid="insertPrice">$1.50</span></td>
    <td><a jwcid="remove">remove</a></td>
  </tr>
  </span>
</table>
```

This example shows a reasonable template, including sample static values used when previewing the HTML layout (they are removed by Tapestry at runtime). Some areas have been glossed over, such as allowing quantities to be changed.

Component `eachItem` is our `Foreach`. It will render its body (all the text and components it wraps) 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

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 `eachItem` 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.

### Example 4.2. Shopping Cart Specification (excerpt)

```
<component id="eachItem" type="Foreach">
  <binding name="source" expression="items"/>
  <binding name="value" expression="lineItem"/>
</component>

<component id="insertProductName type="Insert">
  <binding name="value" expression="lineItem.productName"/>
</component>

<component id="insertQuantity" type="Insert">
  <binding name="value" expression="lineItem.quantity"/>
</component>

<component id="insertPrice" type="Insert">
  <binding name="value" expression="lineItem.price"/>
</component>

<component id="remove" type="ActionLink">
  <binding name="listener" expression="listeners.removeItem"/>
</component>
```

This is very important to the `remove` 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 `ActionLink` 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 from 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). The `ActionLink` component can compare the action id against the target action id encoded within the URL. When a match is found, the `ActionLink` 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, including the use of the `ListEdit` compon-

ent.

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

The application is responsible for providing a listener method, a method which is invoked when the link is triggered.

### Example 4.3. Listener method for remove component

```
public void removeItem(IRequestCycle cycle)
{
  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 the current line item from it. This whole rewinding process has ensured that `lineItem` is the correct value, even though the remove component was rendered several times on the page (because it was wrapped by the `Foreach` component).

### Listener Methods vs. Listener Objects

Listener methods were introduced in Tapestry 1.0.2. Prior to that, it was necessary to create a listener object, typically as an inner class, to be notified when the link or form was triggered. This worked against the basic goal of Tapestry: to eliminate or simplify coding. In reality, the listener objects are still there, they are created automatically and use Java reflection to invoke the correct listener method.

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 shopping cart object (probably stored in the `HttpSession`) and the particular item 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 single, small listener method.

# Stale Links and the Browser Back Button

The fact that web browsers have a "back" button is infuriating to application developers. What right does the user have to dictate the order of navigation through the application? Whose application is this anyway?

In a truly stateless application, the browser back button is not a great hardship, because each page carrys within itself (as cookies, hidden form fields and encoded URLs) all the state necessary to process the page.

Tapestry applications can be more stateful, which is a blessing and a curse. The blessing is that the Tapestry application, running on the server, can maintain state in terms of business objects, data from databases, Enterprise JavaBeans and more. The curse is that a user hitting the back button on the browser loses synchronization with that state.

Let's use an e-commerce example. A user is browsing a list of available cameras from a product catalog. The user clicks on a Minolta camera and is presented with pictures, prices and details about the Minolta camera.

Part of the page lists similar or related items. The user clicks on the name of a similar Nikon camera and is shown the pictures, prices and details of the Nikon camera. The user then hits the browser back button, returning to the page showing the Minolta camera, and clicks the "add to shopping cart" button. Web browsers have no way of informing the server that the user has employed the back button.

Once the user clicks the link, the server replies with a response showing the contents of the shopping cart ... but what has been added to the cart, the Minolta or the Nikon? It depends on how the Tapestry application has been structured.

Presumably, the application has a single page, named `ProductDetails`, that shows the pictures, prices and details of any product. The `ProductDetails` page will have a persistent property named product, of type `Product`. `Product` is a business class that contains all that pricing and detail information.

The question is, how is the add to shopping cart link implemented? If its logic is to add whatever the current value of the product property is (i.e., by using an `ActionLink` component or part of a form) then it will add the Nikon camera, since that's the current product (the most recent one displayed to the user, as far as the server is concerned # it has no way to know the user hit the back button and was staring at the Minolta when the link was clicked). This is the natural approach, since it doesn't take into account the possiblility that the user worked backwards to a prior page.

On the other hand, if a `DirectLink` component is used, it can encode into the URL the primary key of the Minolta product, and that will be the product added to the shopping cart, regardless of the current value of the product property.

HTML Forms, controlled by the `Form` component, are also susceptible to these issues related to the browser back button. Still, there are techniques to make even forms safe. Borrowing an idea from more traditional JavaServer Pages development, a hidden field can be included in the form to sychronize the form and the application ... for example, including the primary key of the Minolta or Nikon product. Tapestry includes a `Hidden` component used for just this purpose.

Finally, the `ListEdit` component exists to help. It works like a `Foreach`, but encodes the number and value of the items it iterates as hidden form fields.

# 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.

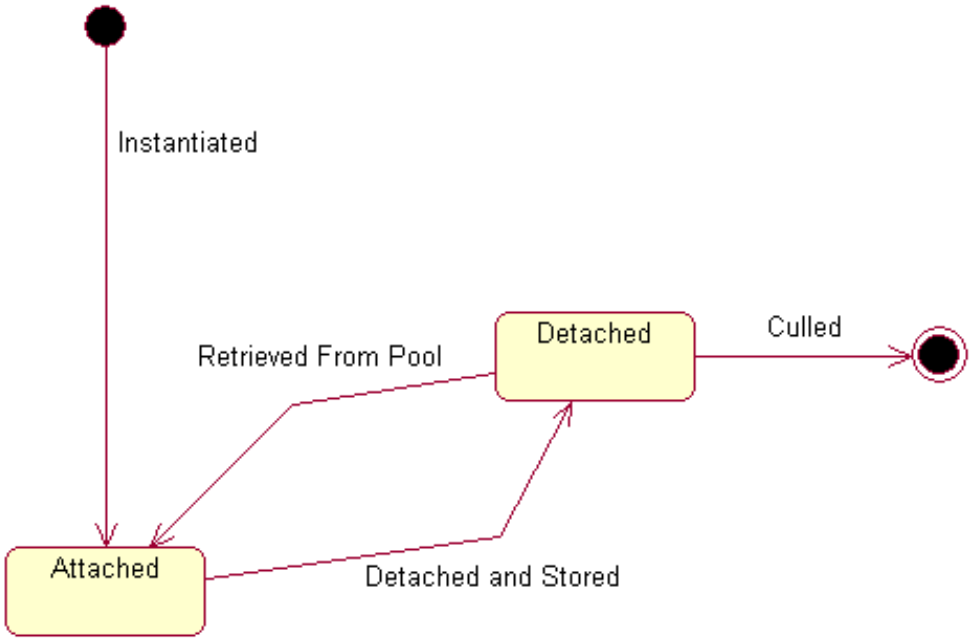
All of this takes time ... not much time on an unloaded server but potentially longer than is acceptable on a busy site.

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 (a single pool is shared by all clients). The maximum number of instances of any one page is determined by the maximum number of clients that simultaneously process a request that involves that page.

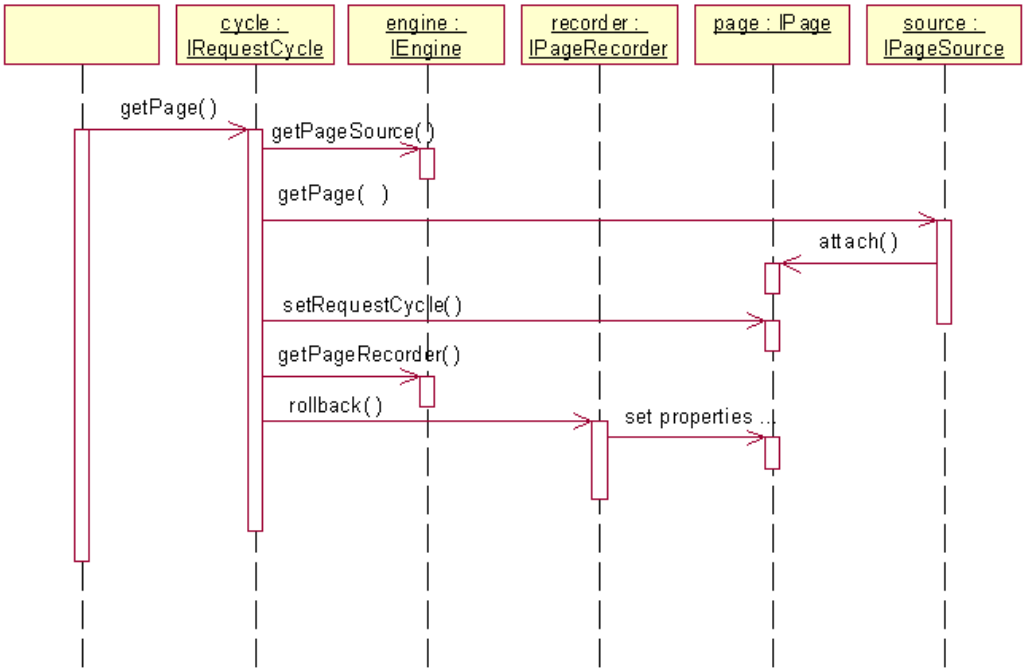
### Figure 4.1. Page Lifecycle




As the page is retrieved from the pool, all of its persistent page properties are set. Thus the page is *equivalent* to the page last used by the application, even if it is not the same instance. This includes any state (that is, the settings of any instance variables) that are particular to the client.

This process is managed by the `IRequestCycle`. When asked for a page, it checks whether the page has been accessed yet for this request. If not, the page must be obtained from the page loader and properly attached and configured.

### Figure 4.2. Page Loading Sequence




The page loader maintains a pool of pages, or can construct a new page instance as needed. The `IPageRecorder` for the page tracks the persistant page properties and can reset the properties of the page instance to values appropriate to the current session.

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 soon as the response HTML stream is sent back to the client, any pages used during the request cycle 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. If, during that window, a second request arrives (from a different client) that involves the same page, a new instance will be created. Unless and until that happens, a single instance will be used and re-used by all clients, regardless of the number of clients.

Pages stay in the pool until culled, at which point the garbage collector will release the memory used by the page (and all the components embedded in it). The default behavior is to cull unused pages after approximately ten minutes.

# Page Localization

When a page is first instantiated, its locale is set to match the locale of the engine 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 engine gets an existing page from the pool, it always matches its locale against the pooled page's locale. Thus a page and

its engine will always agree on locale, with one exception: if the engine locale is changed during the request cycle.

When the engine 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 engine'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.

# Page Buffering

The HTML response generated by a page during rendering is buffered. 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 to the user.

If a page generates a large amount of HTML (larger than the 8KB buffer) 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.

In practice, virtually all Tapestry pages will use a `Body` component wrapping the majority of the page (it takes the place of the normal `<body>` element), and a `Body` component buffers the output of all components in its body. This buffering is necessary so that the `Body` component can write out various JavaScript handlers before the main body of HTML is written (this is often related to the use of the `Rollover` and `Script` components).

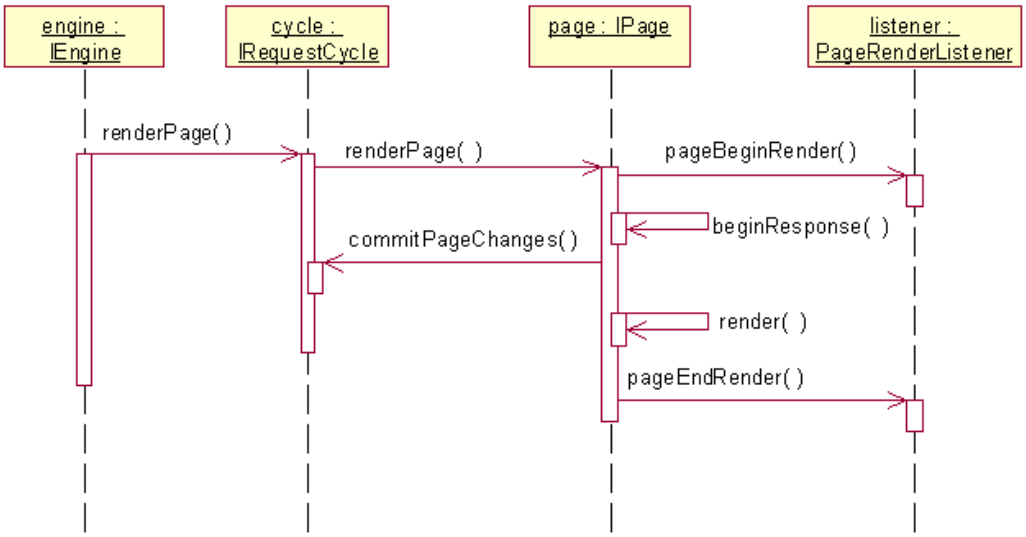
In any case, whenever a `Body` component is used, an exception thrown during the rendering of the page will cause all the HTML buffered by the `Body` component to be cleanly discarded, allowing for a clean presentation of the exception page.

# Page Events

Each page has a lifecycle; it is created and attached to an engine. It will render itself. It is placed in a pool for later reuse. Later, it comes out of the pool and is attached to a new engine to start the process again. There are cases where objects, especially the components embedded somewhere within the page, need to know about this lifecycle.

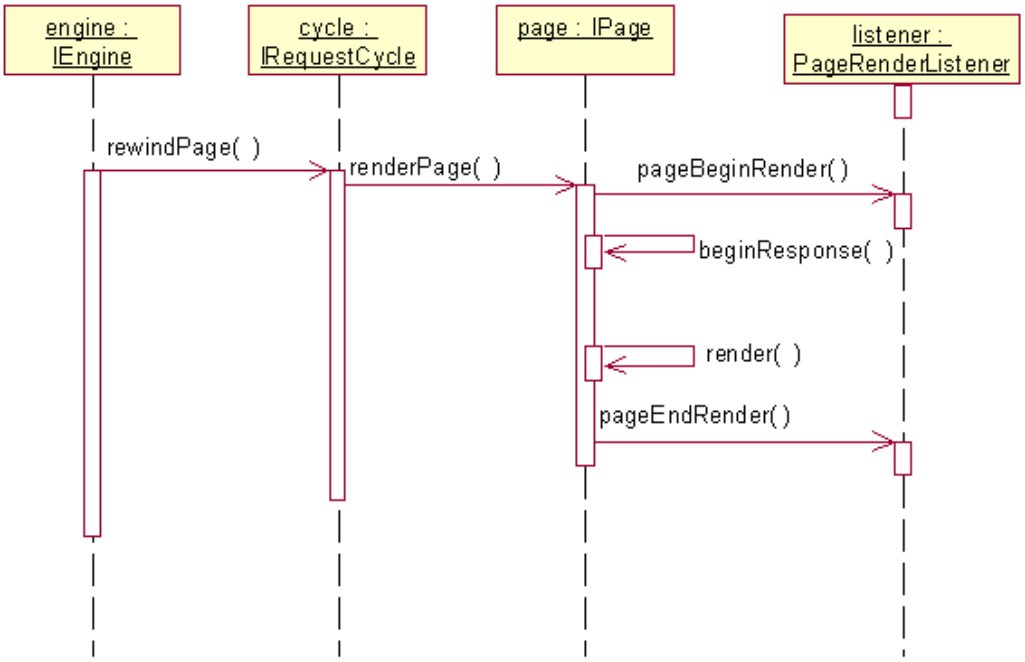
`IPage` can produce a number of events related to its lifecycle. `PageRenderListener` is a listener interface for determining when the page starts and finishes rendering (this includes rewind renders related to the `ActionLink` component).

**Figure 4.3. Page Render Sequence**




The call to `commitPageChanges()` is very important. It is not possible to make any changes to persistant page properties after this method is invoked; doing so will throw an exception.

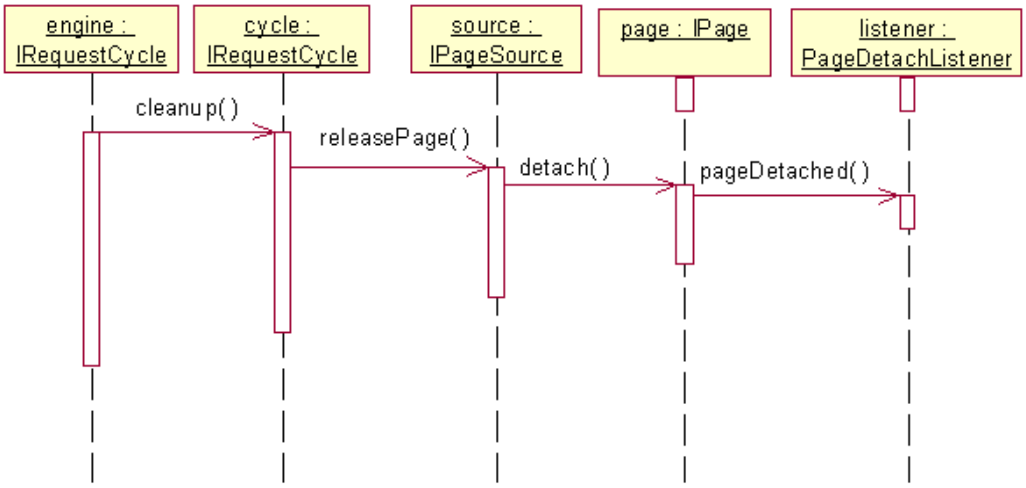
**Figure 4.4. Page Rewind Sequence**




Page rewinds, which are related to the `Form` and `ActionLink` components, also perform a render operation in order to restore dynamic state on the page. The `PageRenderListener` events are still fired. The event listeners can invoke `isRewinding()` on `IRequestCycle` to determine whether this is a normal render, or for rewind purposes.

The `PageDetachListener` interface is used by objects that wish to know when the page is detached from the application, prior to be stored into the page pool (for later reuse). This is used by any components that maintain any independent state.

### Figure 4.5. Page Detach Sequence




This cleanup occurs at the end of the request, after a response has been sent to the client web browser.

The engine knows when the `HttpSession` has been invalidated because the container will invoke `valueUnbound()`. It loads and rolls back each page, then invokes `cleanupPage()` to allow the page to gracefully cleanup any held resources.

Components that implement one of these interfaces usually override the method `finishLoad()` (from `AbstractComponent`) to register themselves with the page.

# Chapter 5. Application Engines and Services

The application engine is a central object whose responsibility is to run the request cycle for each request. To do this, it manages resources such as page loaders and page recorders and provides services to the pages and components utilized during the request cycle.

Application engines are instantiated by the application's servlet (described in the next section). They are stored into the `HttpSession` and are persistent between request cycles.

An important behavior of the engine is to provide named engine services, which are used to create and respond to URLs. The application engine creates and manages the request cycle and provides robust default behavior for catching and reporting exceptions.

The application engine 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 use or subclass `SimpleEngine` 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 engine (which is stored within the `HttpSession`).

# Application Servlet

Every Tapestry application has a single servlet, which acts as a bridge between the servlet container and the application engine. The application servlet is an instance of `ApplicationServlet`.

The first thing a servlet does, upon initialization, is read the application specification. To do this, it must know *where* the application specification is stored.

Specifications are stored on the classpath, which means in a JAR file, or in the `WEB-INF/classes` directory of the WAR.

The servlet determines the location of the application specification from the web deployment descriptor. A servlet initialization property, `org.apache.tapestry.application-specification` provides the locations of the specificiation as a path.

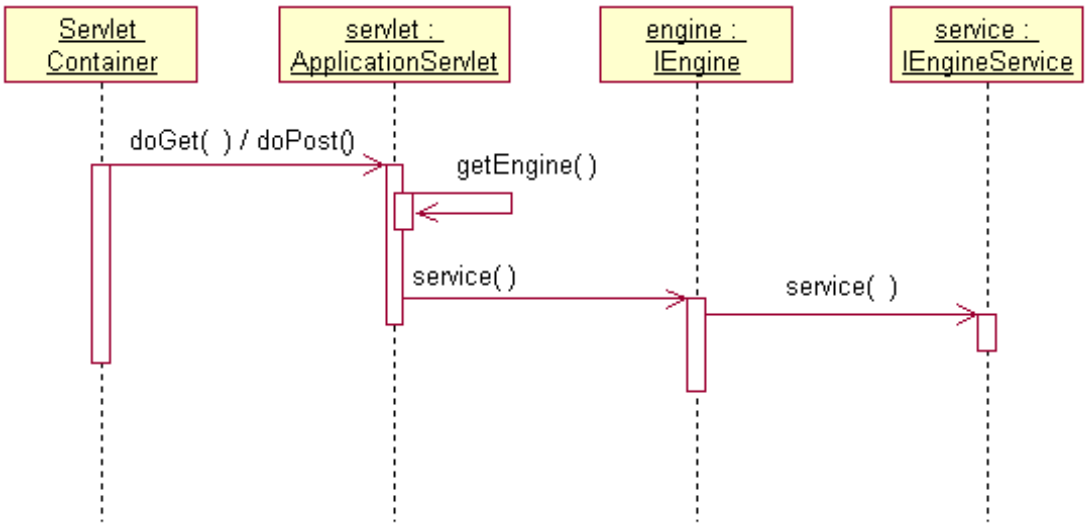
### Example 5.1. Web Deployment Descriptor

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
 "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <display-name>Tapestry Virtual Library Demo</display-name>
  <servlet>
    <servlet-name>vlib</servlet-name>
    <servlet-class>org.apache.tapestry.ApplicationServlet</servlet-class>
    <init-param>
            <param-name>org.apache.tapestry.application-specification</param-name>
            <param-value>/net/sf/tapestry/vlib/Vlib.application</param-value>
    </init-param>
          <load-on-startup>0</load-on-startup>
  </servlet>
```

```
  <!-- The single mapping used for the Virtual Library application -->

  <servlet-mapping>
    <servlet-name>vlib</servlet-name>
    <url-pattern>/app</url-pattern>
  </servlet-mapping>

  <session-config>
          <session-timeout>15</session-timeout>
  </session-config>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

The servlet's main job is to find or create the `IEngine` instance. It then delegates all the behavior for processing the request to the application engine. Encoded in the URL will be a particular application service; the engine delegates to the service to perform the real work of handling the request.

**Figure 5.1. ApplicationServlet Sequence**




# Required Pages

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

**Table 5.1. Tapestry Pages**

| Page Name | Required | Description |
|---|---|---|
| Exception | Default provided, may be overridden. | Page used to present uncaught exceptions to the user. |

| Page Name | Required | Description |
|---|---|---|
| Home | Must be provided by developer. | The initial page displayed when the application is started. |
| Inspector | Provided, never overriden. | Inspector that allows the Tapestry application to be interrogated on its structure. |
| StaleLink | Provided | Page displayed when a StaleLinkException is thrown during the processing of a request. |
| StaleSession | Provided | Page displayed when a StaleSessionException is thrown during the processing of a request. |

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 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 its own `Home` page.

The Exception page is invoked whenever an uncaught 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 `ExceptionAnalyzer` and the `ExceptionDisplay` component are typically used to present this information.

The `StaleLink` page is displayed when a `StaleLinkException` is thrown, which may occur during the processing of the request. The exception is thrown when Tapestry determines that the state of the page (on the server) is out of synch with the client's view of the page ... this most often happens when the user makes use of the browser's back button. [1]

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 create a link back to the `Home` page.

The `StaleSession` page is displayed when a `org.apache.tapestry.StaleSessionException` is thrown. This exception is thrown when the component is configured to be stateful (which is the default) and the `HttpSession` doesn't exist, or is newly created - this indicates a fresh connection to the servlet container after the old session timed out and was discarded. [2]

The `Inspector` page is provided by the framework; it allows a developer to interrogate a running

1 If desired, the application engine can override the method `handleStaleLinkException()`. The default implementation of this method redirects to the `StaleLink` page, but a custom implementation could set up an error message on the application's `Home` page and redirect there instead.

2 Likewise, the default behavior can be changed by overriding the method `handleStaleSessionException()`.

Tapestry application to determine its structure.

# Server-Side State

There are two types of server side state that are supported by Tapestry: persistent page properties and the visit object. The first (page properties) have already been discussed.

The visit object is a central repository for application state and presentation logic. The visit object is accessible through the application engine (the engine implements a `visit` property). The application engine doesn't care about the class of the visit object, or what properties it implements.

The visit object holds central information that is needed by many pages. For example, an e- commerce application may store the shopping cart as a property of the visit object.

When using Enterprise JavaBeans, the visit object is a good place to store remote object references (centralizing the logic to look up home interfaces, instantiate references, etc.).

Every page implements a `visit` property that allows access to the visit object.

When using the `SimpleEngine` engine, the visit object is created the first time it is referenced. The class of the visit object is stored in the application specification.

# Stateful vs. Stateless

Through Tapestry release 1.0.0, an `HttpSession` was created on the very first request cycle, and an engine was created and stored into it.

This comes at some cost, however. Creating the session is somewhat expensive if it is not truly needed, and causes some overhead in a clustering or failover scenario. In fact, until some real server-side state is created; that is, until a persistent page property is recorded or the visit object created, it isn't really necessary to store any server-side state for a particular client.

Starting with Tapestry release 1.0.1, the framework will operate statelessly as long as possible. When triggered (by the creation of a visit, or by a persistent page property) an `HttpSession` will be created and the engine stored within it and the application will continue to operate pretty much as it does in Tapestry release 1.0.0.

While the application continues statelessly, the framework makes use of a pool of engine instances. This is more efficient, as it reduces the number of objects that must be created during the request cycle. However, the major reason for running statelessly is to bypass the overhead statefulness imposes on the application server.

Of course, if rendering the `Home` page of your application triggers the creation of the `HttpSession` [3], then nothing is gained. A well designed application will attempt to defer creation of the session so that, at least, the `Home` page can be displayed without creating a session.

# Engine Services

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

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

3 That is, changes a persistent page property, or forces the creation of the visit object.

Engine 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 a 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 (`ActionLink`, `DirectLink`, `PageLink`). The following example shows how the `PageLink` component is used to create a link between application pages.

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

```
Click <a jwcid="login">here</a> to login.
```

This is combined with the a `<component>` declaration in the the page's specification:

```
<component id="login" type="PageLink">
  <static-binding name="page">Login</static-binding>
</component>
```

The `login` 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, which the `login` component will incorporate into the `<a>` hyperlink it generates.

The resulting HTML:

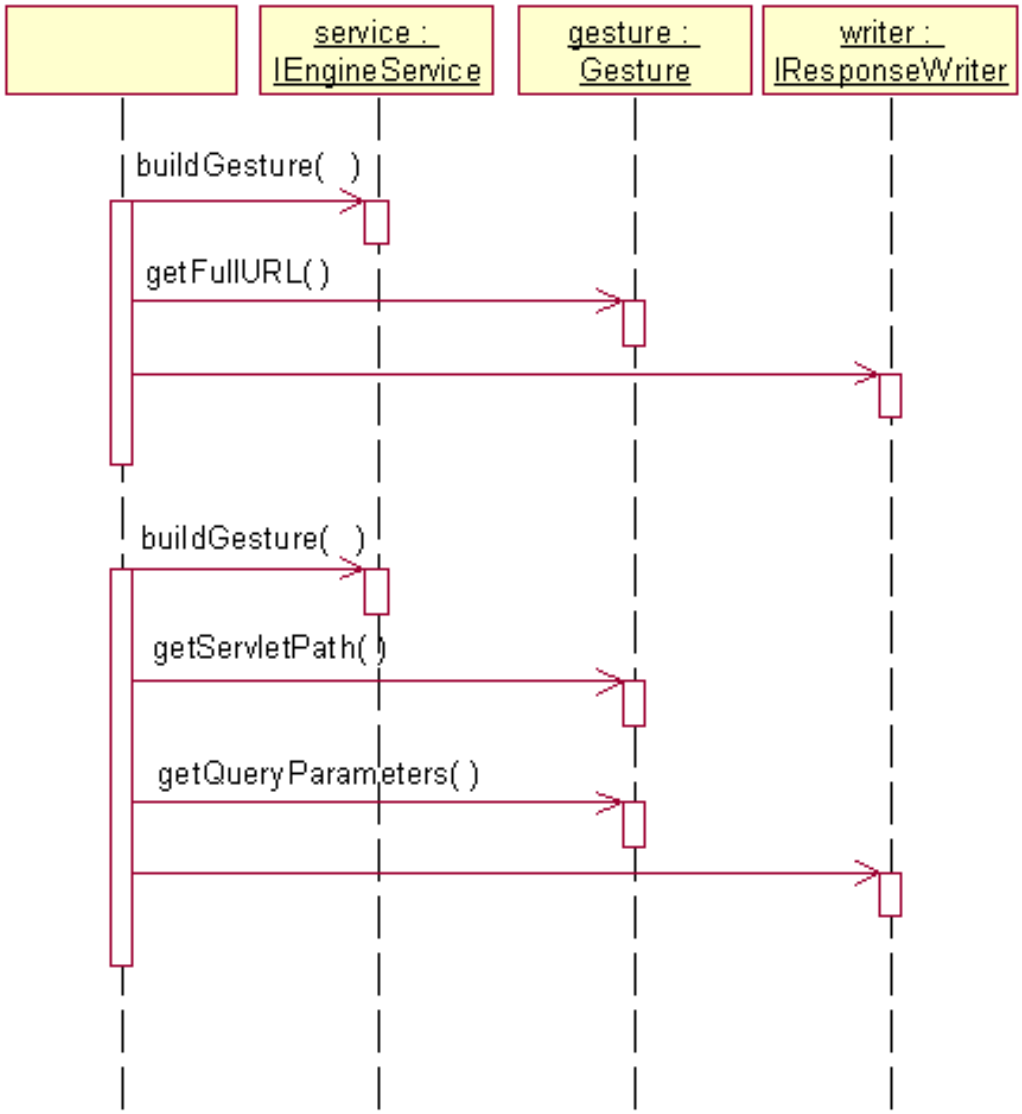
```
Click <a href="/servlet-path?service=page&context=Login">here</a> to login.
```

If the user later clicks that 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.

Links (`ActionLink`, `DirectLink`, etc.) and `Form`s use services in slightly different ways. Links encode all the information directly into the URL whereas `Form`s encode most of the information as hidden form fields.

### Figure 5.2. Services and Gestures




In the first part, a service generates a `Gesture` and then extracts the full URL from it, for use as the `href` attribute of the `<a>` tag.

In the second part, a service is used to access the servlet path (which becomes the `action` attribute of the `<form>` element). The query parameters are individually extracted and encoded as hidden fields in the form.

# Logging

Tapestry makes use of the Apache group's Log4J package to perform logging. This is an easy, fast, powerful framework for adding logging to any Java application. Using Log4J, any number of *loggers* can be created, and a logging level for each logger assigned. Tapestry uses the complete class name as the logger for each class.

The `ApplicationServlet` class includes a method, `setupLogging()`, to help initialize Log4J, allowing the default configuration to be overridden using command line parameters.

The Tapestry Inspector includes a Logging tab that allows the logging configuration to be dynamically changed. The logging level for any logger can be assigned, and new loggers can be created.

What this means is that, using the Inspector, it is possible to control exactly what logging output is produced, dynamically, while the application is still running. The Tapestry Inspector is easily added to any Tapestry application.

# Private Assets

The application engine is responsible for making private assets, assets that are stored on the Java classpath, visible when necessary to client web browser.

This takes two forms:

- Dynamic download of asset data via the application servlet.
- Dynamic copying of asset data into the web server's virtual file system.

The first form is the default behavior; each private asset requires an additional round trip through the application server and application engine to retrieve the stream of bytes which make up the asset. This is fine during development, but less than ideal at deployment, since it places an extra burden on the servlet container, stealing valuable cycles away from the main aspects of servicing end users.

The second form is better during deployment. The bytestreams are copied out of the classpath to a specific directory, one that is mapped into the web server's virtual file system. Once it is so copied, the access to the asset is completely static, as with any other image file or HTML page.

To enable dynamic copying, it is necessary to inform the framework about what file system directory to copy the assets to, and what virtual file system directory that maps to. This is accomplished using a pair of JVM system properties:

### JVM System Properties

`org.apache.tapestry.asset.dir`
The complete pathname of a directory to which private assets may be copied by the asset externalizer.

`org.apache.tapestry.asset.URL`
The URL corresponding to the external asset directory.

# 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-volumes systems 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 engine, 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 consecutive request cycles. In the first request cycle, a particular page is rendered and, along the way, any number of URLs are generated and included in the HTML. The second request cycle is triggered by one of those service URLs.

## Service URLs and query parameters

All URLs generated by the framework consist of the the path to the servlet, and up to three query parameters.

- `service`: the name of the service that will be used to processes the request.
- `context`: contextual information needed by the service; typically the name of the page or component involved. Often there are several pieces of information, separated by slashes.
- `sp`: additional parameters that can be made available to the component. This is used by a `DirectLink` component. If there is more than one service parameter, then there will be multiple `sp` parameters in the URL.

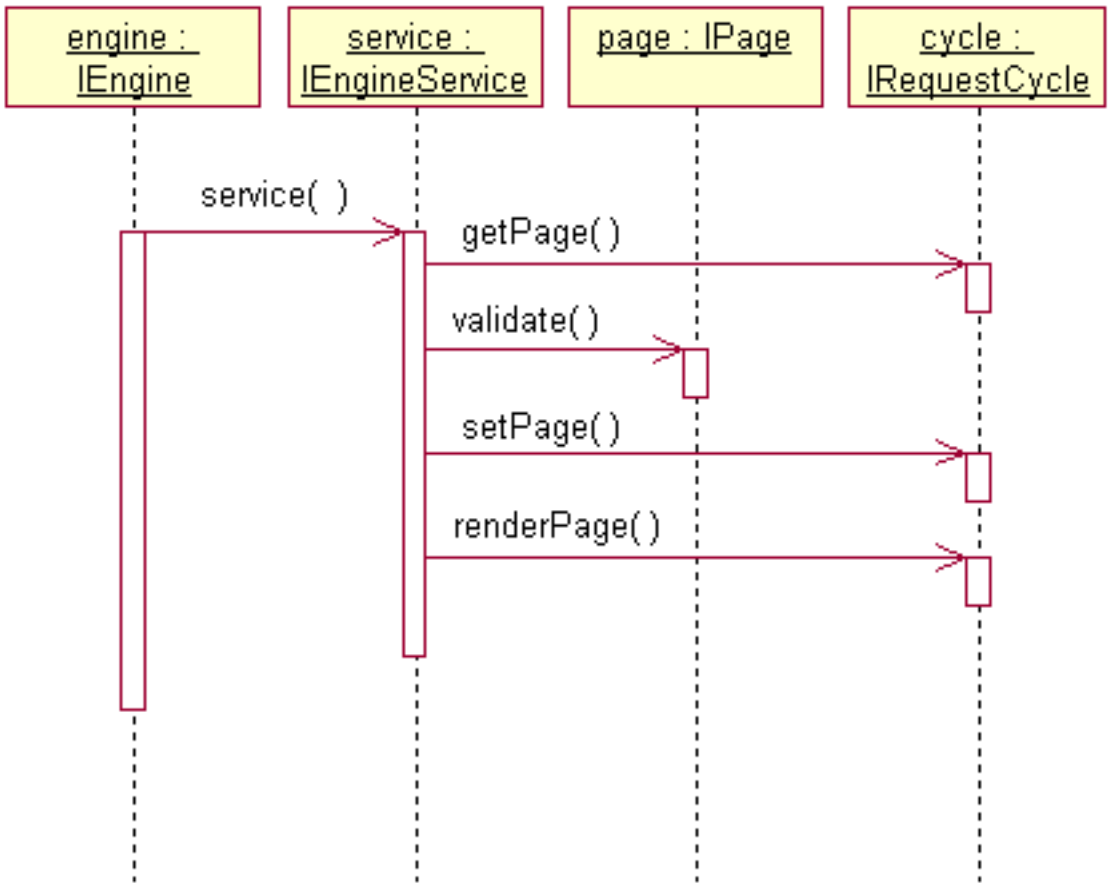
## Page service

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

A page service stores the name of the page as the single value in the service context.

The request cycle for the page service is relatively simple.

**Figure 6.1. Page Service Sequence**




The URL contains the name of the page, and the corresponding page is aquired from the request cycle. The page is given a chance to validate that the user can access it, it can throw `PageRedirectException` to force a render of a different page. Otherwise, `setPage()` tells the request cycle which page will be used to render a response, and `renderPage()` peforms the actual render.

# Action and Direct listeners

The `ActionLink`, `DirectLink` and `Form` components (which make use of the action and direct services) inform the application when they have been triggered using listeners.

A listener is an object that implements the `IActionListener` interface.

```
public void actionTriggered(IComponent component, IRequestCycle cycle)
  throws RequestCycleException;
```

Prior to release 1.0.2, it was necessary to create an object to be notified by the component; this was almost always an annonymous inner class:

```
public IActionListener getFormListener()
{
  return new IActionListener()
  {
    public void actionTriggered(IComponent component, IRequestCycle cycle)
      throws RequestCycleException
    {
      // perform some operation ...
    }
  };
}
```

Although elegant in theory, that's simply too much Java code for too little effect. Starting with Tapestry 1.0.2, it is possible to create a *listener method* instead.

A listener method takes the form:

```
public void method-name(IRequestCycle cycle)
throws RequestCycleException;
```

## Note

The throws clause is optional and may be omitted. However, no other exception may be thrown.

In reality, listener *objects* have not gone away. Instead, there's a mechanism whereby a listener object is created automatically when needed. Each component includes a property, `listeners`, that is a collection of listener objects for the component, synthesized from the available public methods. The listener objects are properties, with the names corresponding to the method names.

## Tip

The class `AbstractEngine` (the base class for `SimpleEngine`) also implements a listeners property. This allows you to easily add listener methods to your application engine.

The earlier example is much simpler:

```
public void formSubmit(IRequestCycle cycle)
{
  // perform some operation ...
}
```

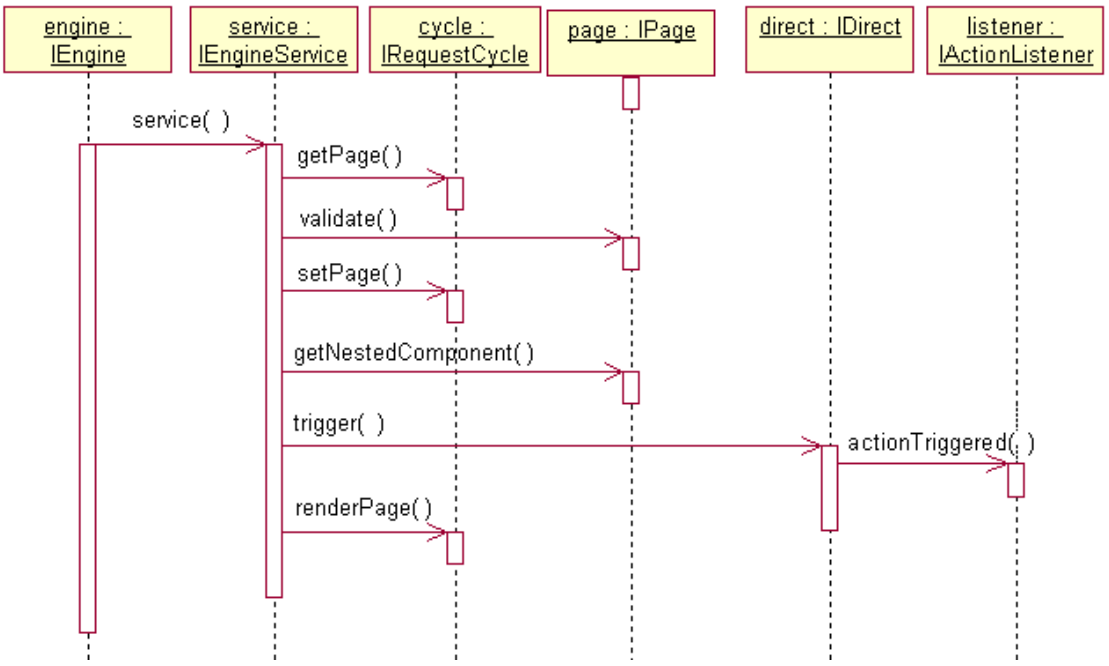
However, the property path for the listener binding must be changed, from `formListener` to `listeners.formSubmit`.

# Direct service

The direct service is used to trigger a particular action. This service is tied to the `DirectLink` component. The service context identifies the page and component within the page. Any parameters specified by the `DirectLink` component's `context` parameter are encoded as well.

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

**Figure 6.2. Direct Service Sequence**




As with the page service, the page involved has a chance validate the request. The component is located within the page, and the page is set as the default response page. The listener is free to override this, and can load other pages, change their properties, or otherwise affect the state of the application.

After the listener has its chance to respond to the request, a response page is renderred.

### `IDirect` vs. `DirectLink`

The sequence shown above is for the `DirectLink` component, which implements the `IDirect` interface. In some rare cases, it is desirable to have a different component implement the `IDirect` interface instead. It will still implement the `trigger()` method, but will respond in its own way, likely without a listener.

This is the primary way (besides forms) in which applications respond to the user. What's key is the component's listener, of type `IActionListener`. This is the hook that allows pre-defined cotheirponents from the Tapestry framework to access application specific behavior. The page or container of the `DirectLink` component provides 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 `ActionLink` or `Form` components) should be used.

The `DirectLink` component has an optional parameter named `parameters`. The value for this may be a single object, an array of objects, or a `List`. Each object is converted into a string encoding, that is included in the URL. When the action is triggered, the array is reconstructed (from the URL) and stored in the `IRequestCycle`, where it is available to the listener. The type is maintained, thus if the third parameter is of type `Integer` when the URL is generated, then the third parameter will still be an `Integer` when the listener method is invoked.

This is a very powerful feature of Tapestry, as it allows the developer to encode dynamic page state directly into the URL when doing so is not compatible with the action service (described in the next section).

The most common use for these service parameters is to record an identifier for some object that is affected by the link. For example, if the link is designed to remove an item from the shopping cart (in an e-commerce example), the service parameters could identify which item to remove in terms of a primary key, or line number within the order.

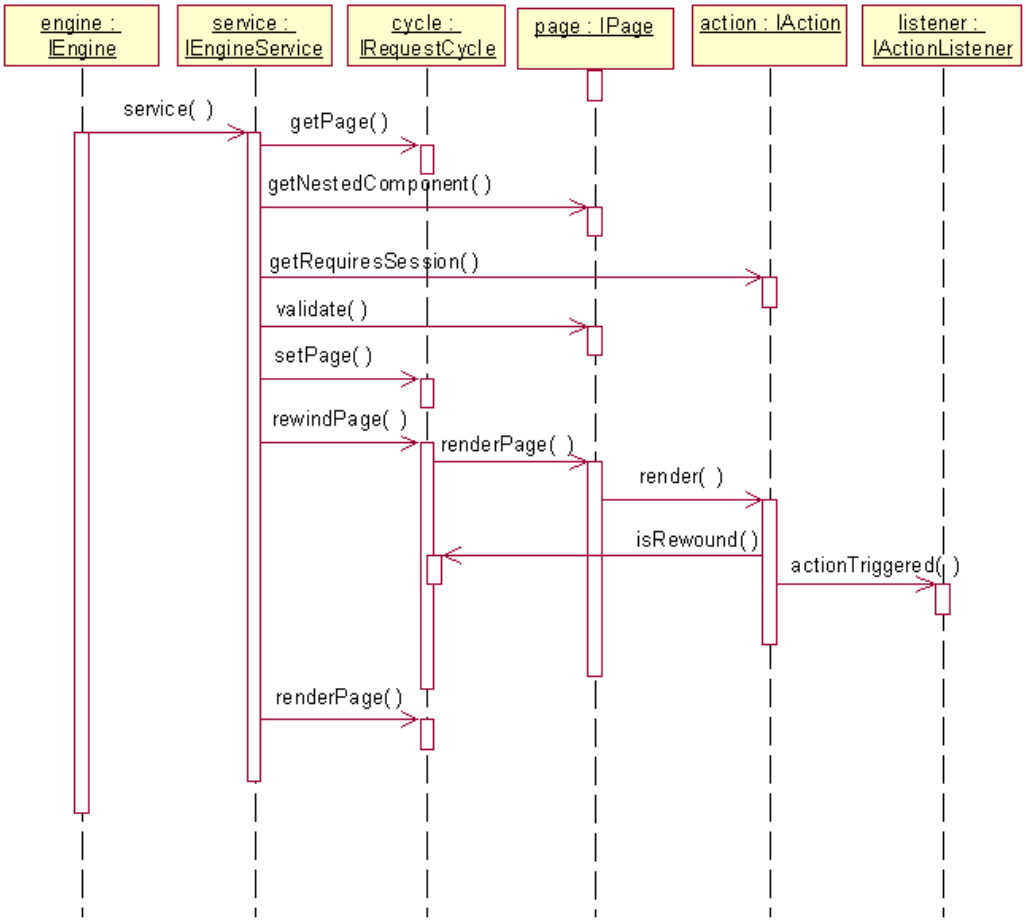
# Action service

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

An action service encodes the page name and component for the request. It also includes an action id.

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

**Figure 6.3. Action Service Sequence**




The point of the action service is to restore the *dynamic state* of the page to how it was when the `ActionLink` component rendered the link. Only then is the listener notified.

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 `ActionLink` component'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 `DirectLink` component) instead, as it allows the necessary context to be encoded into the URL, using service parameters. 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 the service parameters, to create links to a product details page.

# Services and forms

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

In order to accept the results posted in the HTML form, each of these components must be given a chance to 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.

As with an `ActionLink` component, a full rewind must be done, to account for conditional portions of the page and any `Foreach` components.

## Note

Starting with Tapestry release 1.0.2, `Form`s may now use the direct service instead of the action service; this is configurable. Using the direct service is the default behavior unless specified. A rewind still occurs, it simply starts directly with the `Form` component, rather than having to "work down" to it. This can be a performance gain if a page contains many forms.

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, `TextArea` and `TextField`, are quite simple. They simply move text between the application, the HTML and the submitted request.

Individual `Checkbox` components are also simple: they set a boolean property. A `RadioGroup` and some `Radio` components allow a property to be set to a value (dependent on which radio button is selected by the user). The `PropertySelection` component is designed to more efficiently handle this and can produce HTML for either a popup list or a collection of radio buttons.

Tapestry also includes the more involved component, `ValidField`, which is similar to the simple `TextField` component, but provide greater validation and checking of input, and provides the ability to visually mark fields that are required or in error.

Regardless of which service the `Form` uses, it encodes the query parameters (which identify the service and context) as hidden field elements, rather than encoding them into the URL. This is necessary because some servlet containers ignore URL query parameters when using the HTTP POST request; therefore, it is necessary that all query parameters (including the ones related to the engine service), be part of the form posting ... and that means the use of hidden fields in the form.

# Chapter 7. 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. Each implemention of the `IEngine` interface provides a different strategy.

Currently, only the `SimpleEngine` class is provided with the framework; it uses in-memory page recorders. When the engine 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 engine 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 engine 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.

The Tapestry Inspector can be used to monitor the size of the serialized engine in a running application.

## 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. Having the wizard name match the package name is also helpful.

The designer must also account for additional entry points to the application beyond the standard home page. These may 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 visit object. The visit object may implement methods for adding products to the shopping cart. This could take the form of Java methods, component listeners.

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

Most Tapestry applications will involve some interaction with Enterprise JavaBeans. The code to lookup home interfaces, or to gain access to ession beans, is typically located in the visit object.

Listener code, on various pages, will cast the visit object to the appropriate actual class and invoke methods.

The following example demonstrates this idea. Visit is a hypothetical visit object that uses EJBs.

```
public void exampleListener(IRequestCycle cycle)
{
  Visit visit; ❶
  ISomeHomeInterface home;

  visit = (Visit)getVisit();

  home = visit.getSomeHomeInterface();

  try
  {
          // etc.
  }
  catch (RemoteException ex)
  {
    throw new ApplicationRuntimeException(ex);
  }
}
```

❶ Each application can freely define the type of the visit object, and its is common to call the class "Visit". Another option is to create a subclass for the engine and store home interfaces there.

# Identify Engine Services

Tapestry applications will need to define new engine 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 bookmarked, but when the bookmark is triggered, the page may not render correctly, because it will not know which product to display. 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 bookmarked,

that context is lost.

By defining a new engine service, the necessary context can be encoded directly into the URL, in a way 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 service URL could encode some form of product identifier.

An example of this is in the Virtual Library application. In order to make certain pages bookmarkable, a new service named "external" was created.

The external service includes the name of a page and the primary key of an object that page displays (this was simplified by the design of the Vlib entity beans, which always use an `Integer` as the primary key).

The external service works much the same was as the page service, except that it invokes an additional method on the page, `setup()`, which is passed the primary key extracted from the URL.

The end result is that when a user arrives at such a page, the URL used identifies the page and the primary key. Bookmarking the page stores the URL so that when the bookmark is later opened, the correct data is read and displayed.

# 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).

Other examples of common components are viewers and editors of common data types.

In the Virtual Library, components that make use of the external service were created. The components, `BookLink` and `PersonLink`, took as parameters the corresponding objects (`Book` or `Person`) and created links to the pages that displayed the details of that `Book` or `Person`.

# Chapter 8. Coding Tapestry Applications

After performing the design steps from the previous chapter, it is time to start coding. The designs will imply certain requirements for the implementations.

## Application Engine

Application engines 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 engine can locate the specification from its servlet (the servlet reads the application specification once, when it is first initialized).

This is largely not an issue, since most applications use a provided class, such as `SimpleEngine`. Subclassing is only necessary when the application needs a different method of instantiating the visit object, or needs to store additional data (see Operating Stateless). In some cases, it is convienient to create a subclass to provide common component listener methods.

## Visit Object

The visit object will contain all the data about a client's visit to the web application. If possible, it should have a no-arguments constructor (this allows `SimpleEngine` to instantiate it as needed).

Keeping the size of the serialized engine small is a good goal for overall performance and scalability, and the visit object is serialized with the engine. During initial development, the visit object should implement the `java.io.Serializable` interface.

Once the application, and the structure of the visit object, is stable, the more efficient `java.io.Externalizable` interface should be implemented instead.

In addition, deferring the creation of the visit object as late as possible is also of benefit, since this is the best way to keep the serialized engine small.

## Operating Stateless

Tapestry applications can operate in a stateless mode, that is, without a `HttpSession`. The framework automatically creates a session when needed; when the Visit object is first created, or when any persistent page properties are changed.

Ideally, the `Home` page of the application should not trigger the creation of a session: it should be careful not to create the Visit object. Remember that hits on your application will form a curve: The `Home` page is at the top of the curve, and it drops of rapidly as users penetrate deeper into the application ... how many times have you visited the front page of a web site and gone no further?

Stateless operations will affect `ActionLink`, `DirectLink` and `Form` components on your pages. By default, they will reject requests while the application is running stateless; the user will be redirected to the `StaleSession` page. This is appropriate, since normally, the lack of a session means that the previous session timed out and was discarded.

Each of these components has a `stateful` parameter which may be bound to `false`. When `stateful` is false, the components will accept stateless requests.

As the developer, you must keep a careful eye on what's stateful vs. stateless, and look to move stateless data into the engine, so as to avoid creating a visit object as long as possible. For example, the engine can resolve and store EJB home interfaces and references to *stateless* session EJBs. Even read-only database data can be stored in the engine. However, anything that is related to a particular user must be stored in the visit object (or a persistent page property).

It is also important to not accidentally create the visit object. Every page includes a `visit` property which will create the visit if it doesn't already exist. This will, in turn, force the creation of an `HttpSession`. On the other hand, the property path `engine.visit` will *not* create the visit object. To avoid creating the visit, you may need to wrap some of your HTML template inside a `Conditional` component whose condition parameter is bound to the property `engine.visit`.

# Enterprise JavaBeans Support

The visit object should provide access to the most commonly used Enterprise JavaBeans used in the application. It can provide a central location for the common code (related to JNDI and to narrowing EJB references), rather than have that scattered throughout the application.

It is important to remember that EJB references are not serializable. However, it is possible to convert between an EJB reference and an EJB handle, and handles are serializable. The visit should make any instance variables that store EJB references transient, and should perform extra serialization work to serialize and restore the necessary EJB handles.

Also remember that persistent page properties that are EJB references are *automatically* converted to handles when stored, and back into references when restored.

# Page classes

It is often useful to create one or two subclasses of `BasePage` specific to your application. Often your application will have a consistent navigational border on some or all pages that can be supported by the base class. Many applications have one set of pages that are visible to unidentified guests, and a second section that is visible once the user logs in. A base class for the second set of pages could override the `validate()` method to redirect to a login page if the user is not already logged in.

# Chapter 9. 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 state. A component which does should implement the `PageDetachListener` interface, and implement the `pageDetached()` method.

The `pageDetached()` method is invoked from the page's `detatch()` 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 `AbstractComponent`. Such components must implement the protected `renderComponent()` method.

Components that use templates inherit from a subclass of `AbstractComponent`: `BaseComponent`. They should leave the `renderComponent()` method alone.

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 without subclassing. This is even more possible when using helper beans.

# 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.

### Connected Parameters

Most components use "in" parameters and can have Tapestry connect the parameters to properties of the component automatically. This discussion reveals some inner workings of Tapestry that developers most often no longer need to be aware of.

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 by name 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

`renderComponent()` method is below:

```
protected void renderComponent(IMarkupWriter writer, IRequestCycle cycle)
  throws RequestCycleException
{
  IBinding colorBinding = getBinding("color");
  Color color = (Color)colorBinding.getObject("color", Color.class);
  String 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 `Map` and then casting the result from `java.lang.Object` to `org.apache.tapestry.IBinding`.

Because bindings are typically set once and then 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" of type `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;
}


protected void renderComponent(IMarkupWriter writer, IRequestCycle cycle)
  throws RequestCycleException
{
  Color color = (Color)colorBinding.getObject("color", Color.class);
  String 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` to store the binding will never be created.

# Persistent Component State

As with pages, 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.

In addition, the component should implement the interface `PageDetachListener`, and implement the method `pageDetached()`, and, within that method, reset all instance variables to default values, just as a page does (in its `detach()` 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, context and private.

- An external asset is just a fancy way of packaging a URL at an arbitrary web site.
- A context asset represents a file with a URL relative to the web server containing the Tapestry application.
- A private asset is a file within the classpath, 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. For example, the standard exception page makes use of a private asset to access its stylesheet.

In a traditional web application, private assets would need to be packaged separately from the 'compon-

ent' 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 classpath.

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 classpath and to a directory visible to the web server. This requires some additional configuration.
- The assets are dynamically accessed from the class path using the asset service.

Copying assets out of the classpath 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 `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'.

# Chapter 10. Tapestry and JavaScript

Building cutting edge Web applications is not entirely about the server side. A significant amount of work must be done on the client side to support truly dynamic user experiences. Typically, this scripting is done using the JavaScript language embedded into major web browsers such as Internet Explorer and Netscape Navigator.

These effects range from simple effects such as image rollovers (changing the icon used for a link when the cursor is over it) to more involved patterns such as client side validation of forms or even complex animations.

In traditional, static web page development, the HTML producer (the person creating the static HTML page) is completely responsible for this aspect of development, usually aided by a web page authoring tool, such as Dreamweaver. Ultimately, though, the HTML producer assigns unique names or ids to various elements on the page, and attaches JavaScript event handlers to the elements.

### Example 10.1. Traditional JavaScript usage

```
var preload = new Array();
preload[0] = new Image();
preload[0].src = "/images/button.gif";
preload[1] = new Image();
preload[1].src = "/images/button-highlight.gif";

function rollover(image, index)
{
  image.src = preload[index].src;
}

.
.
.
  <a href="..."
    onMouseOver="javascript:rollover(document.button, 1);"
    onMouseOut="javascript:rollover(document.button, 0);">
    <img name="button" src="/images/button.gif">
  </a>
```

The preloading business is all about forcing the browser to load the image *before* it is needed, so that it is already in memory when the mouseover event handler needs it.

From here, adding additional rollovers means extending the `preload` array, providing names for the additional `<img>` elements and writing the additional event handlers for the `<a>` elements.

Now, envision a running Tapestry application. With everything so dynamic (especially when you account for things like the `Foreach` component), it's all but impossible to even know how many links and buttons will be on the page, never mind what they'll all be named. At first glance, it may appear that Tapestry prevents the use of this kind of scripting.

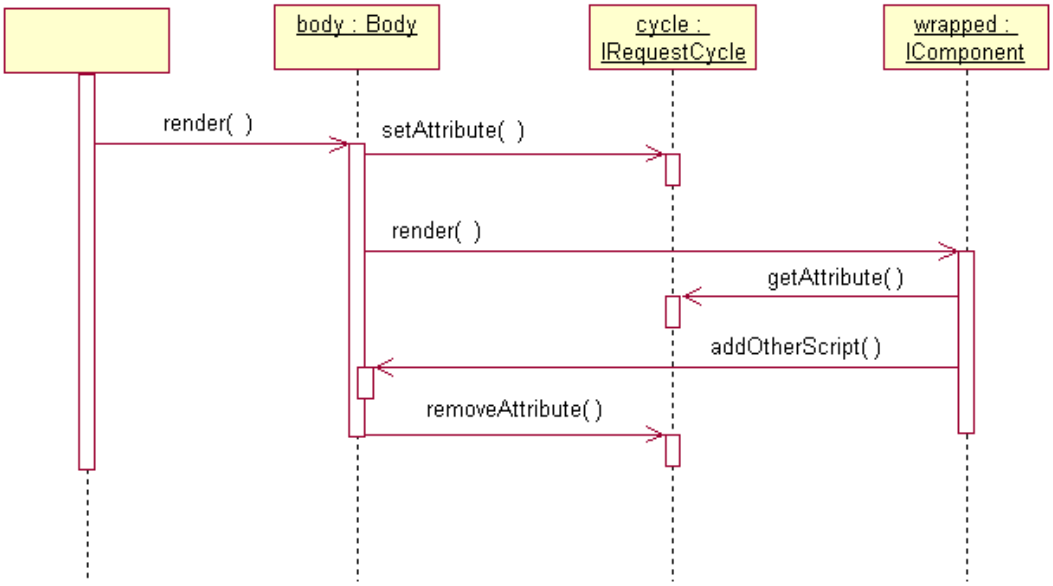
In fact, Tapestry is structured to enhance this kind of scripting. This is faciliated by the `Body` component, which replaces the `<body>` element of the page. The next section described the services the `Body` component povides to facilitate complex client-side scripting.

# The Body component

The `Body` component provides a number of services to the components it wraps. It handles preloading of images. It provides the ability to add arbitrary JavaScript to the page, to include an external static JavaScript document, or to add JavaScript to the `<body>` element's onload event handler. Finally, it provides an easy way to generate unique identifiers needed for things like image and function names.

When the `Body` component renders, it registers itself as an attribute of the `IRequestCycle`. This allows components wrapped by the `Body` component, directly or indirectly, to locate it and invoke methods on it. These methods are used to define preloaded images, and add JavaScript code to the response HTML.

**Figure 10.1. Body Component Rendering Sequence**




When rendering is complete, the `Body` component will have produced four distinct portions of the HTML response:

```
<script language="JavaScript" src="..."></script> ❶
<script language="JavaScript"><!-- ❷

...

function tapestry_onLoad()  ❸
{
}

// --> </script>
<body onload="javascript:tapestry_onLoad();"> ❹

... ❺

</body>
```

❶ Any number of included static scripts may be added to the page.
❷ This script block is only emitted when necessary; that is, because some component needed to generate scripting or initialization (or preloaded images). The block is properly "commented" so that older browsers, those that fail to support scripting, will not be confused by the JavaScript code.
❸ The onload event handler function is only generated if some component requests some onload initialization.
❹ The `<body>` tag only specfies a `onload` event handler function if one is needed.
❺ The content of the `<body>` element is defined by the Tapestry components it wraps. Importantly, the rollovers, JavaScript, event handlers and the content are all generated in parallel (the `Body` component uses buffering so that the JavaScript portion is written out first).

# Script Specifications and Script Components

The `Body` component only lays the foundation for client-side JavaScript support in Tapestry. Tapestry includes its own, XML-based language for create dynamic JavaScript.

A Tapestry Script Specification takes as input a number of *symbols*, each of which is a named object. These input symbols are combined to form additional symbols. Additional XML tags allow a script to place JavaScript into the main script body, or into the initialization.

The most common use for script specifications is to add client-side behavior to form elements. The input symbol is a form component, from this, the name of the element and containing form are determined. Next, the name of one or more event handler functions are defined.

In the body, the functions are actually created. In the initialization, the event handlers are wired to the form and form elements.

In some cases, a script specification may produce usable output symbols (commonly, the names of a JavaScript function that should be tied to some component's event handler).

## Note

A detailed example is coming.

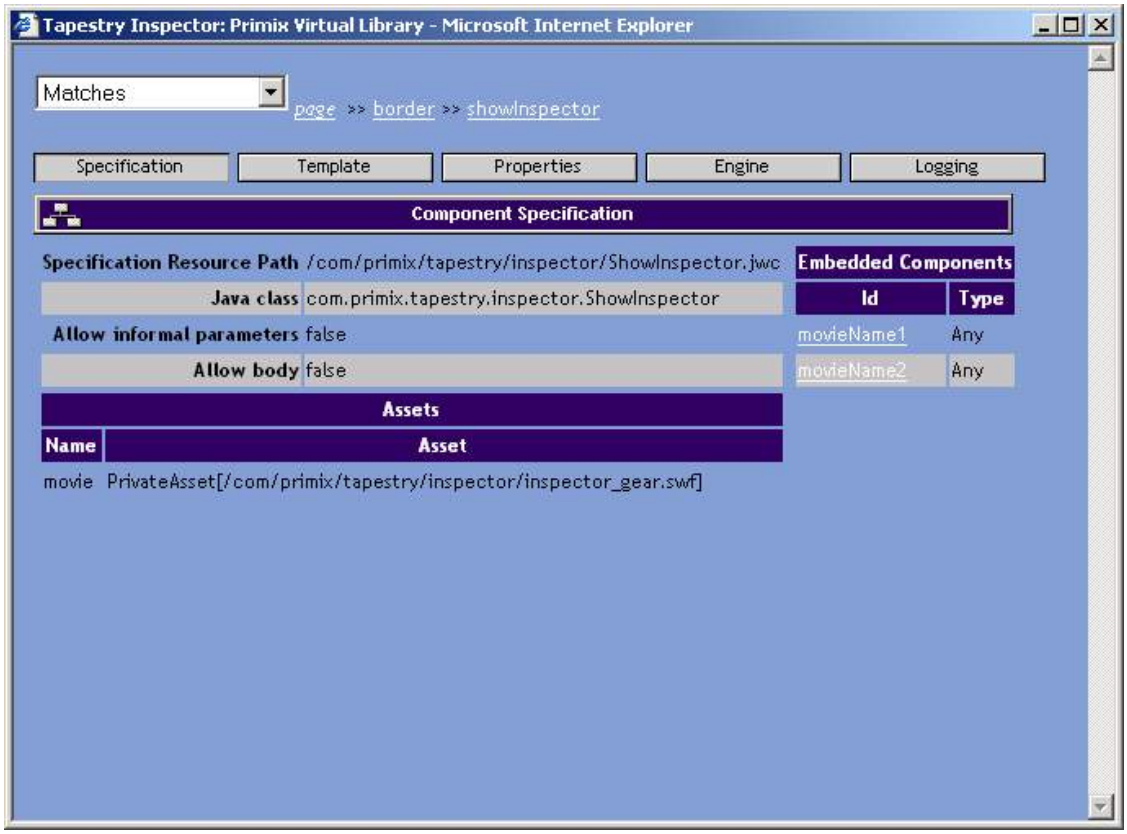
# Chapter 11. The Tapestry Inspector

Tapestry includes a powerful tool: the Inspector, which reveals the construction of a running Tapestry application.

The `InspectorButton` component is used to include a link that launches the Inspector. This is typically included in the navigational border of an application, so that it is available to the developer from any page. The Inspector itself is a page that is provided by the framework and available to any Tapestry application.

## Specification View

**Figure 11.1. Inspector - Specification View**

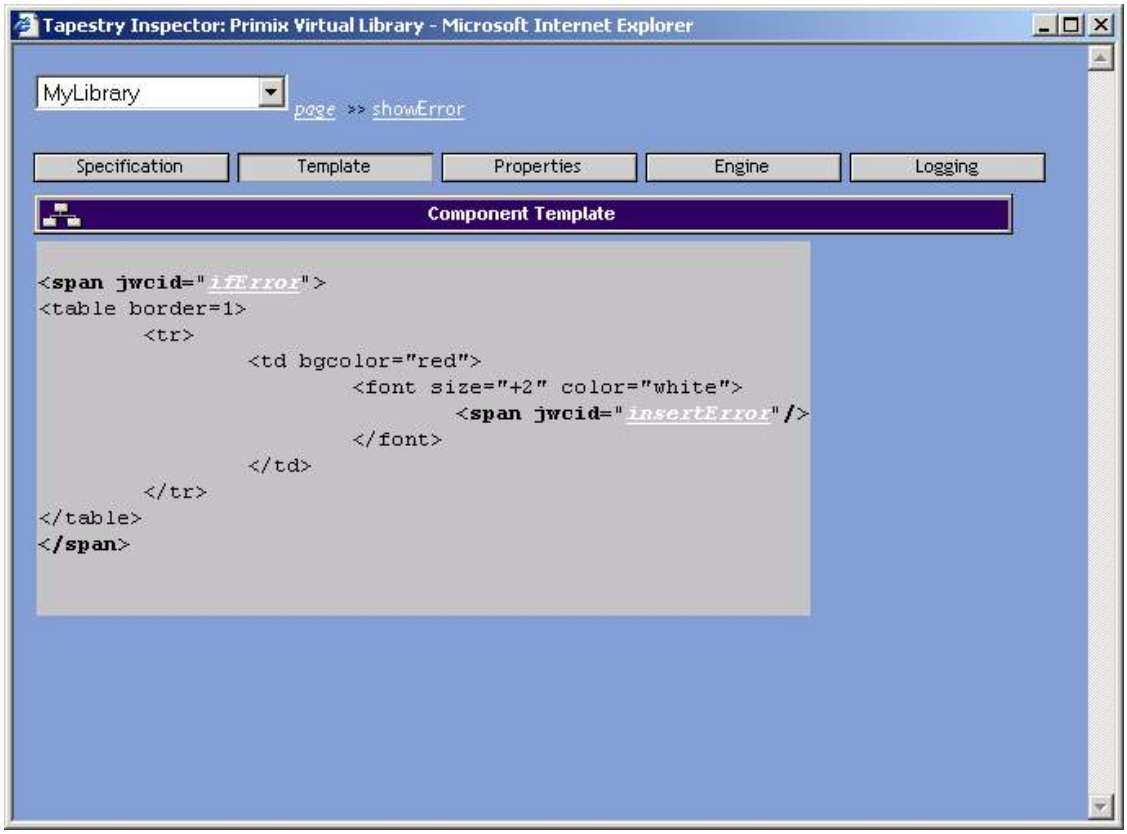



The Inspector allows the developer to see how the page is constructed. It reveals the page's specification, a list of embedded components within the page, the page's HTML template and more.

It is possible to dig down and see the same information for any component within the page.

## Template View

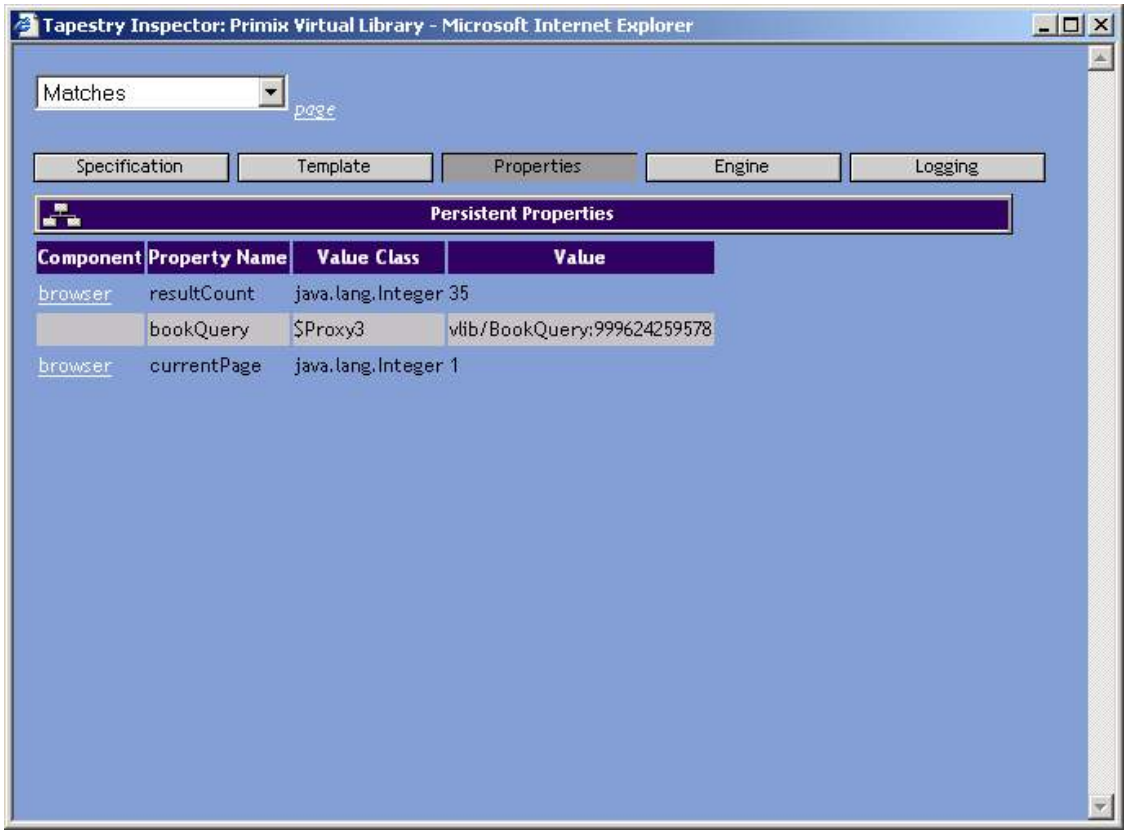
**Figure 11.2. Inspector - Template View**




The template view shows the HTML template for the component. Within the template, component references are links that "dig down" into their template (if they have one).

# Properties View

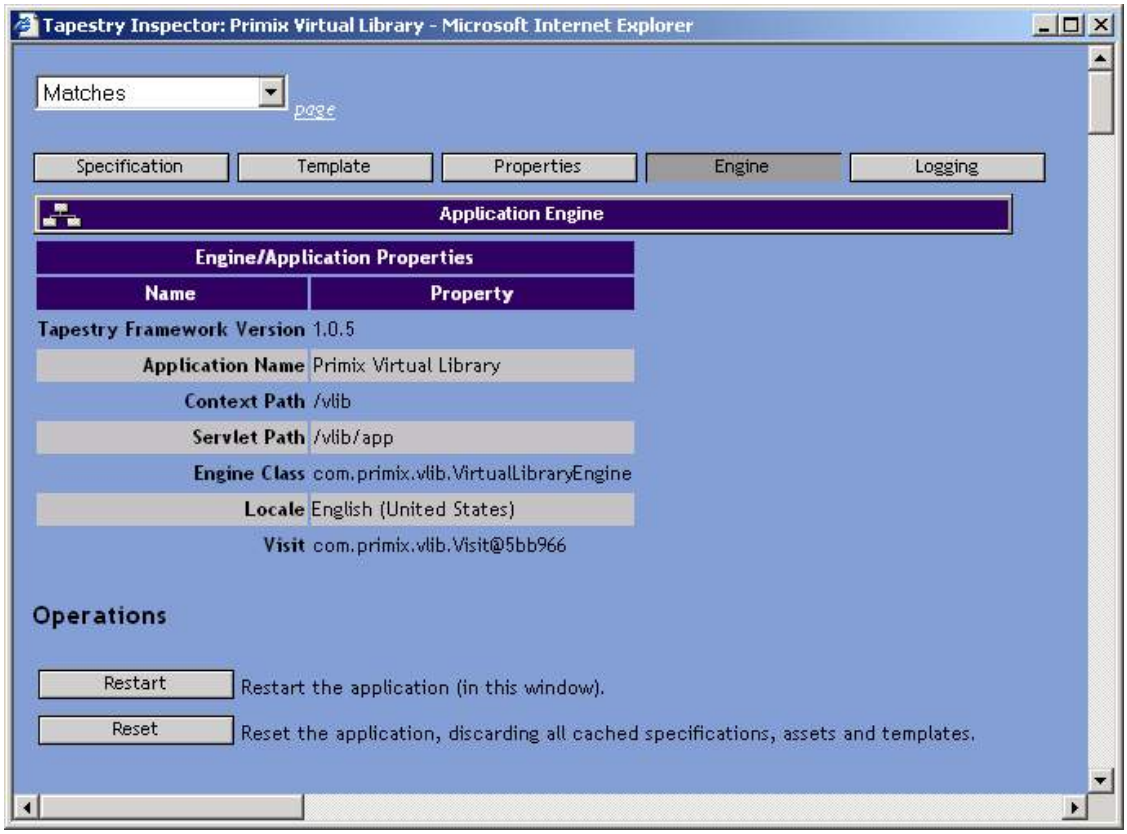
**Figure 11.3. Inspector - Properties View**




The properties view shows the persistent properties for the page and any components on the page.

# Engine View

**Figure 11.4. Inspector - Engine View**

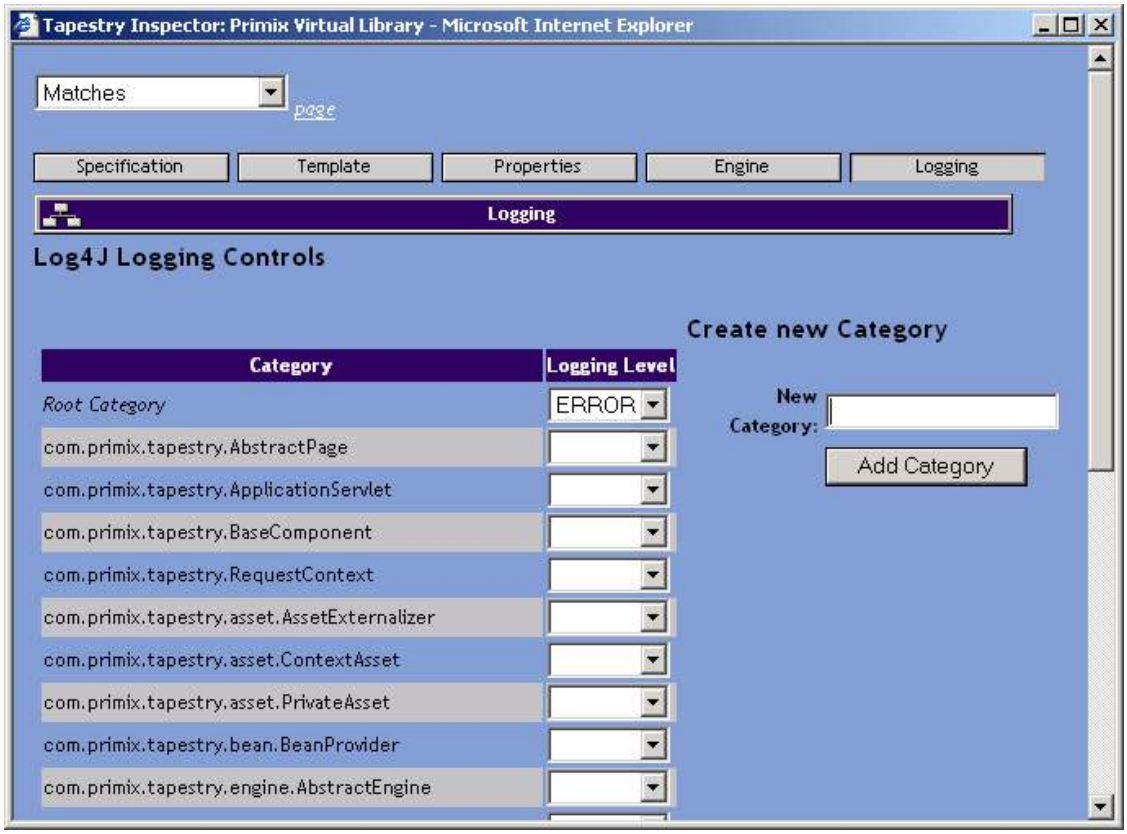



The engine view shows information about the running engine instance, including its name and class.

Not shown in the figure is the serialized state of the application engine (in a hex dump format) and a long display of all the request cycle information (the same information produced when an uncaught exception is thrown).

# Logging View

**Figure 11.5. Inspector - Logging View**




The final tab allows control of the logging behavior of the application. It allows the logging level for any category to be set, and allows new categories to be created.

# Appendix A. Tapestry JAR files

`lib/runtime/*.jar`

Frameworks that are usually needed at runtime (but not at framework build time) and are not always supplied by the servlet container. This currently is just the Log4J framework.

`lib/ext/*.jar`

Frameworks needed when compiling the framework and at runtime. This is several other Jakarta frameworks (including BSF and BCEL), plus the OGNL framework.

`tapestry-3.0.jar`

The main Tapestry framework. This is needed at compile time and runtime. At runtime, it is most often added to the servlet container's classpath. The framework release number is integrated into the file name.

`tapestry-contrib-3.0.jar`

Contains additional components and tools that are not integral to the framework itself, such as the `Palette`. Needed at runtime if any such components are used in an application. The framework release number is integrated into the file name.

In addition, Tapestry applications may need the packages
<class>javax.servlet</class>
and
<class>javax.xml.</class>
at compile time and an XML parser at runtime. These are usually provided by the servlet container or application server.

# Appendix B. Tapestry Specification DTDs

This appendix describes the four types of specifications used in Tapestry.

**Table B.1. Tapestry Specifications**

| Type | File Extension | Root Element | Public ID | System ID |
|---|---|---|---|---|
| Application | application | `<application>` | `-//Howard Lewis Ship//Tapestry Specification 1.3//EN` | `http://tapestry.sf.net/dtd/Tapestry_1_3.dtd` |
| Page | page | `<page-specification>` | `-//Howard Lewis Ship//Tapestry Specification 1.3//EN` | `http://tapestry.sf.net/dtd/Tapestry_1_3.dtd` |
| Component | jwc | `<component-specification>` | `-//Howard Lewis Ship//Tapestry Specification 1.3//EN` | `http://tapestry.sf.net/dtd/Tapestry_1_3.dtd` |
| Library | library | `<library-specification>` | `-//Howard Lewis Ship//Tapestry Specification 1.3//EN` | `http://tapestry.sf.net/dtd/Tapestry_1_3.dtd` |
| Script | script | `<script>` | `-//Howard Lewis Ship//Tapestry Script 1.2//EN` | `http://tapestry.sf.net/dtd/Tapestry_1_2.dtd` |

The four general Tapestry specifications (`<application>`, `<component-specification>` `<page-specification>` and `<library-specification>`) all share the same DTD, but use different root elements.

## `<application>` element

*root element*

The application specification defines the pages and components specific to a single Tapestry application. It also defines any libraries that are used within the application.

**Figure B.1. `<application>` Attributes**

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | User presentable name of application. |
| engine-class | string | yes | | Name of an implementation of IEngine to instantiate. |

**Figure B.2. `<application>` Elements**

```
<description>*, <property>*,
(<page>|<component-alias>|<service>|<library>|<extension>)*
```

# <bean> element

Appears in: `<component-specification>` and `<page-specification>`

A `<bean>` is used to add behaviors to a page or component via aggregation. Each `<bean>` defines a named JavaBean that is instantiated on demand. Beans are accessed through the OGNL expression `beans.`*`name`*.

Once a bean is instantiated and initialized, it will be retained by the page or component for some period of time, specified by the bean's lifecycle.

### bean lifecycle

none
The bean is not retained, a new bean will be created on each access.

page
The bean is retained for the lifecycle of the page itself.

render
The bean is retained until the current render operation completes. This will discard the bean when a page or form finishes rewinding.

request
The bean is retained until the end of the current request.

Caution should be taken when using lifeycle `page`. A bean is associated with a particular instance of a page within a particular JVM. Consecutive requests may be processed using different instances of the page, possibly in different JVMs (if the application is operating in a clustered environment). No state particular to a single client session should be stored in a page.

Beans must be public classes with a default (no arguments) constructor. Properties of the bean may be configured using the `<set-property>` and `<set-string-property>` elements.

**Figure B.3. <bean> Attributes**

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the bean, which must be a valid Java identifier. |
| class | string | yes | | The name of the class to instantiate. |
| lifecycle | none\|page\|renderer\|request | no | request | As described above; duration that bean is retained. |

**Figure B.4. <bean> Elements**

```
<description>*, <property>*,
(<set-property>|<set-string-property>)*
```

# <binding> element

Appears in: <component>

Binds a parameter of an embedded component to an OGNL expression rooted in its container.

In an instantiated component, bindings can be accessed with the OGNL expression bindings.*name*.

**Figure B.5. <binding> Attributes**

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the parameter to bind. |
| expression | string | yes | | The OGNL expression, relative to the container, to be bound to the parameter. |

# <configure> element

Appears in: <extension>

Allows a JavaBeans property of the extension to be set from a statically defined value. The <configure> element wraps around the static value. The value is trimmed of leading and trailing

whitespace and optionally converted to a specified type before being assigned to the property.

### Figure B.6. `<configure>` Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| property-name | string | yes | | The name of the extension property to configure. |
| type | `boolean\|int\|long\|double\|String` | no | `String` | The conversion to apply to the value. |

# `<component>` element

Appears in: `<component-specification>` and `<page-specification>`

Defines an embedded component within a container (a page or another component).

In an instantiated component, embedded components can be accessed with the OGNL expression `components.`*`id`*.

### Figure B.7. `<component>` Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| id | string | yes | | Identifier for the component here and in the component's template. Must be a valid Java identifier. |
| type | string | no | | A component type to instantiate. |
| copy-of | string | no | | The name of a previously defined component. The type and bindings of that component will be copied to this component. |

Either `type` or `copy-of` must be specified.

A component type is either a simple name or a qualified name. A simple name is the name of an component either provided by the framework, or provided by the application (if the page or component is defined in an application), or provided by the library (if the page or component is defined in a library).

A qualified name is a library id, a colon, and a simple name of a component provided by the named library (for example, `contrib:Palette`). Library ids are defined by a `<library>` element in the containing library or application.

**Figure B.8. `<component>` Elements**

```
<property>*,
(<binding>|<field-binding>|<inherited-binding>|<static-binding>|<string-binding>)
```

# `<component-alias>` element

Appears in: `<application>` and `<library-specification>`

Defines a component type that may latter be used in a `<component>` element (for pages and components also defined by this application or library).

**Figure B.9. `<component-alias>` Attributes**

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| type | string | yes | | A name to be used as a component type. |
| specification-path | string | yes | | The complete resource path to the component's specification (including leading slash and file extension). |

# `<component-specification>` element

*root element*

Defines a new component, in terms of its API (`<parameter>`s), embedded components, beans and assets.

The structure of a `<component-specification>` is very similar to a `<page-specification>` except components have additional attributes and elements related to parameters.

**Figure B.10. `<component-specification>` Attributes**

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| class | string | yes | | The Java class to instantiate, which must implement the interface `IComponent`. Typically, this is `BaseComponent` or |

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| | | | | a subclass of it. |
| allow-body | `yes\|no` | no | `yes` | If `yes`, then any body for this component, from its containing page or component's template, is retained and may be produced using a `Render-Body` component.<br><br>If `no`, then any body for this component is discarded. |
| allow-inform-al-parameters | `yes\|no` | no | `yes` | If `yes`, then any informal parameters (bindings that don't match a formal parameter) specified here, or in the component's tag within its container's template, are retained. Typically, they are converted into additional HTML attributes.<br><br>If `no`, then informal parameters are not allowed in the specification, and discarded if in the template. |

**Figure B.11. `<component-specification>` Elements**

```
<description>*, <parameter>*, <reserved-parameter>*, <property>*,
(<bean>|<component>|<external-asset>|<context-asset>|<private-asset>)*
```

# `<context-asset>` element

Specifies an asset located relative to the web application context root folder. Context assets may be localized.

Assets for an instantiated component (or page) may be accessed using the OGNL expression `assets.`*`name`*.

**Figure B.12. `<context-asset>` Attributes**

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the asset, which must be a valid Java identifier. |
| path | string | yes | | The path to the asset, assuming a root directory matching the servlet context root directory. The path should begin with leading forward slash. |

# `<description>` element

Appears in: *many*

A description may be attached to a many different elements. Descriptions are used by an intelligent IDE to provide help. The Tapestry Inspector may also display a description.

## Note

The DTD supports multiple `<description>` elements, each localized to a different language. In practice, a single description, in English, is typically used. This approach to providing a localized description is likely to change in the future, and it is probably safest to expect just a single `<description>` tag to be allowed in the next revision of the DTD.

The descriptive text appears inside the `<description>` tags. Leading and trailing whitespace is removed and interior whitespace may be altered or removed. Descriptions should be short; external documentation can provide greater details.

**Figure B.13. `<description>` element**

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| xml:lang | string | no | | The language the message is localized to as an ISO language string. |

# `<extension>` element

Appears in: `<application>` and `<library-specification>`

Defines an extension, a JavaBean that is instantiated as needed to provide a global service to the application.

### Figure B.14. `<extension>` Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | A name for the extension, which can (and should) look like a qualified class name, but may also include the dash character. |
| class | string | yes | | The Java class to instantiate. The class must have a zero-arguments constructor. |
| immediate | `yes\|no` | no | `no` | If `yes`, the extension is instantiated when the specification is read. If `no`, then the extension is not created until first needed. |

### Figure B.15. `<component-specification>` Elements

```
<property> *, <configure> *
```

# `<external-asset>` element

Appears in: `<component-specification>` and `<page-specification>`

Defines an asset at an arbitrary URL. The URL may begin with a slash to indicate an asset on the same web server as the application, or may be a complete URL to an arbitrary location on the Internet.

External assets may be accessed at runtime with the OGNL expression `assets.`*`name`*.

### Figure B.16. `<external-asset>` Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | A name for the asset. Asset names must be valid Java identifiers. |
| URL | string | yes | | The URL used to ac- |

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| | | | | cess the asset. |

# <field-binding> element

Appears in: <component>

Binds a parameter of an embedded component to a public static final field.

## Note

Although the same result can be accomplished using a <binding> element and the OGNL expression @*class-name*@*field-name*, using a <field-binding> is more efficient, because Tapestry knows that the value is invariant.

The class name must be the qualified class name. If the package is ommitted, java.lang is assumed (this makes it easier to reference common fields such as Boolean.TRUE).

In an instantiated component, bindings can be accessed with the OGNL expression bindings.*name*.

### Figure B.17. <field-binding> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the parameter to bind. |
| field-name | string | yes | | The name of a public static final field, in the form *class-name.field-name*. |

# <inherited-binding> element

Appears in: <component>

Binds a parameter of an embedded component to a parameter of its container.

In an instantiated component, bindings can be accessed with the OGNL expression bindings.*name*.

### Figure B.18. <inherited-binding> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the parameter to bind. |
| parameter-name | string | yes | | The name of a para- |

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| | | | | meter of the containing component. |

# <library> element

Appears in: <application> and <library-specification>

Establishes that the containing application or library uses components defined in another library, and sets the prefix used to reference those components.

### Figure B.19. <library> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| id | string | yes | | The id associated with the library. Components within the library can be referenced with the component type *id*:*name*. |
| specification-path | string | yes | | The complete resource path for the library specification. |

# <library-specification> element

*root element*

Defines the pages, components, services and libraries used by a library. Very similar to <application>, but without attributes related application name or engine class.

The <library-specification> element has no attributes.

### Figure B.20. <library-specification> Elements

```
<description>*, <property>*,
(<page>|<component-alias>|<service>|<library>|<extension>)*
```

# <page> element

Appears in: <application> and <library-specification>

Defines a page within an application (or contributed by a library). Relates a logical name for the page to the path to the page's specification file.

### Figure B.21. <page> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name for the page, which must start with a letter, and may contain letters, numbers, underscores and the dash character. |
| specification-path | string | yes | | The complete resource path to the page's specification. |

# <page-specification> element

*root element*

Defines a page within an application (or a library). The <page-specification> is a subset of <component-specification> with attributes and entities related to parameters removed.

### Figure B.22. <page-specification> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| class | string | yes | | The Java class to instantiate, which must implement the interface IPage. Typically, this is BasePage or a subclass of it. |

### Figure B.23. <page-specification> Elements

```
<description> *, <property> *,
(<bean>|<component>|<external-asset>|<context-asset>|<private-asset>)*
```

# <parameter> element

Appears in: `<component-specification>`

Defines a formal parameter of a component.

### Figure B.24. `<parameter>` Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the parameter, which must be a valid Java identifier. |
| java-type | scalar name, or class name | no | | Required for connected parameters. Specifies the type of the JavaBean property that a connected parameter writes and reads. The property must match this exact value, which can be a fully specified class name, or the name of a scalar Java type. |
| required | `yes\|no` | no | `no` | If `yes`, then the parameter must be bound (though it is possible that the binding's value will still be null). |
| property-name | string | no | | For connected parameters only; allows the name of the property to differ from the name of the parameter. If not specified, the property name will be the same as the parameter name. |
| direction | `in\|form\|custom` | no | `custom` | Identifies the semantics of how the parameter is used by the component. `custom`, the default, means the component explicitly controls reading and writing values through the binding.<br><br>`in` means the property is set from the parameter before the component renders, and is reset back to |

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| | | | | default value after the component renders.<br><br>`form` means that the property is set from the parameter when the component renders (as with `in`). When the form is submitted, the value is read from the property and used to set the binding value after the component rewinds. |

# `<private-asset>` element

Specifies located from the classpath. These exist to support reusable components packages (as part of a `<library-specification>`) packaged in a JAR. Private assets will be localized.

Assets for an instantiated component (or page) may be accessed using the OGNL expression `assets.`*`name`*.

### Figure B.25. `<private-asset>` Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the asset, which must be a valid Java identifier. |
| resource-path | string | yes | | The path to the asset on the classpath. The path should begin with leading forward slash. |

# `<property>` element

Appears in: *many*

The `<property>` element is used to store meta-data about some other element (it is contained within). Tapestr ignores this meta-data Any number of name/value pairs may be stored. The value is the static text the `<property>` tag wraps around.

### Figure B.26. `<property>` Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the property. |

# <reserved-parameter> element

Appears in: <component-specification>

Used in components that allow informal parameters to limit the possible informal parameters (so that there aren't conflicts with HTML attributes generated by the component).

All formal parameters are automatically reserved.

Comparisons are caseless, so an informal parameter of "SRC", "sRc", etc., will match a reserved parameter named "src" (or any variation), and be excluded.

### Figure B.27. <reserved-parameter> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the reserved parameter. |

# <service> element

Appears in: <application> and <library-specification>

Defines an IEngineService provided by the application or by a library.

The framework provides several services (home, direct, action, external, etc.). Applications may override these services by defining different services with the same names.

Libraries that provide services should use a qualified name (that is, put a package prefix in front of the name) to avoid name collisions.

### Figure B.28. <service> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the service. |
| class | string | yes | | The complete class name to instantiate. The class must have a zero-arguments constructor and implement the interface IEngineService |

# <set-property> element

Appears in: <bean>

Allows a property of a helper bean to be set to an OGNL expression (evaluated on the containing component or page).

### Figure B.29. <set-property> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the helper bean property to set. |
| expression | string | yes | | The OGNL expression used to set the property. |

# <set-string-property> element

Appears in: <bean>

Allows a property of a helper bean to be set to a localized string value of its containing page or component.

### Figure B.30. <set-string-property> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the helper bean property to set. |
| key | string | yes | | A string property key of the containing page or component. |

# <static-binding> element

Appears in: <component>

Binds a parameter of an embedded component to a static value. The value, which is stored as a string, is the wrapped contents of the <static-binding> tag. Leading and trailing whitespace is removed.

In an instantiated component, bindings can be accessed with the OGNL expression bindings.*name*.

### Figure B.31. <static-binding> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the parameter to bind. |

# `<string-binding>` element

Appears in: `<component>`

Binds a parameter of an embedded component to a localized string of its containing page or component.

In an instantiated component, bindings can be accessed with the OGNL expression `bindings.`*`name`*.

### Figure B.32. `<string-binding>` Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| name | string | yes | | The name of the parameter to bind. |
| key | string | yes | | The localized property key to retrieve. |

# Appendix C. Tapestry Script Specification DTD

Tapestry Script Specifications are frequently used with the `Script` component, to create dynamic JavaScript functions, typically for use as event handlers for client-side logic.

The root element is `<script>`.

A script specifcation is a kind of specialized template that takes some number of input symbols and combines and manipulates them to form output symbols, as well as body and initialization. Symbols may be simple strings, but are also frequently objects or components.

Script specifications use an Ant-like syntax to insert dynamic values into text blocks. `${`*`OGNL expression`*`}`. The expression is evaluated relative to a `Map` of symbols.

## <body> element

Appears in: `<script>`

Specifies the main body of the JavaScript; this is where JavaScript variables and methods are typically declared. This body will be passed to the `Body` component for inclusion in the page.

**Figure C.1. <body> Elements**

```
(text|<foreach>|<if>|<if-not>)*
```

## <foreach> element

Appears in: *many*

An element that renders its body repeatedly, much like a `Foreach` component. An expression supplies a collection or array of objects, and its body is rendered for each element in the collection.

**Figure C.2. <foreach> Attributes**

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| key | string | yes | | The symbol to be updated with each successive value. |
| expression | string | yes | | The OGNL expression which provides the source of elements. |

### Figure C.3. `<foreach>` Elements

```
(text|<foreach>|<if>|<if-not>)*
```

# `<if>` element

Appears in: *many*

Conditionally renders its body, if a supplied OGNL expression is true.

### Figure C.4. `<if>` Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| expression | string | yes | | The OGNL expression to be evaluated. |

### Figure C.5. `<if>` Elements

```
(text|<foreach>|<if>|<if-not>)*
```

# `<if-not>` element

Appears in: *many*

Conditionally renders its body, if a supplied OGNL expression is false.

### Figure C.6. `<if-not>` Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| expression | string | yes | | The OGNL expression to be evaluated. |

### Figure C.7. `<if-not>` Elements

```
(text|<foreach>|<if>|<if-not>)*
```

# <include-script> element

Appears in: <script>

Used to include a static JavaScript library. A library will only be included once, regardless of how many different scripts reference it. Such libraries are located on the classpath.

### Figure C.8. <include-script> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| resource-path | string | yes | | The location of the JavaScript library. |

# <initialization> element

Appears in: <script>

Defines initialization needed by the remainder of the script. Such initialization is placed inside a method invoked from the HTML <body> element's onload event handler ... that is, whatever is placed inside this element will not be executed until the entire page is loaded.

### Figure C.9. <initialization> Elements

```
(text|<foreach>|<if>|<if-not>)*
```

# <input-symbol> element

Appears in: <script>

Defines an input symbol for the script. Input symbols can be thought of as parameters to the script. As the script executes, it uses the input symbols to create new output symbols, redefine input symbols (not a recommended practice) and define the body and initialization.

This element allows the script to make input symbols required and to restrict their type. Invalid input symbols (missing when required, or not of the correct type) will result in runtime exceptions.

### Figure C.10. <input-symbol> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| key | string | yes | | The input symbol to be checked. |
| class | string | no | | If specified, this is the |

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| | | | | complete, qualified class name for the symbol. The provided symbol must be assignable to this class (be a subclass, or implement the specified class if the specified class is actually an interface). |
| required | `yes \| no` | no | `no` | If `yes`, then a non-null value must be specified for the symbol. |

# `<let>` element

Appears in: `<script>`

Used to define (or redefine) a symbol. The symbol's value is taken from the body of element (with leading and trailing whitespace removed).

### Figure C.11. `<let>` Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| key | string | yes | | The key of the symbol to define. |

### Figure C.12. `<let>` Elements

```
(text|<foreach>|<if>|<if-not>)*
```

# `<script>` element

*Root element*

The root element of a Tapestry script specification.

### Figure C.13. `<script>` Elements

```
<include-script> *, <input-symbol> *,
(<let> | <set>) *,
<body> ?, <initialization> ?
```

# <set> element

Appears in: <script>

A different way to define a new symbol, or redefine an existing one. The new symbol is defined using an OGNL expression.

### Figure C.14. <set> Attributes

| Name | Type | Required ? | Default Value | Description |
|---|---|---|---|---|
| key | string | yes | | The key of the symbol to define. |
| expression | string | yes | | The OGNL expression to evaluate. |