Tapestry User's Guide

by Howard Lewis Ship

Tapestry User's Guide

by Howard Lewis Ship

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1. Introduction

Tapestry is a component-based web application framework, written in Java. Tapestry is more than a simple templating system; Tapestry builds on the Java Servlet API to build a platform for creating dynamic, interactive web sites. More than just another templating language, Tapestry is a real framework for building complex applications from simple, reusable components. Tapestry offloads much of the error-prone work in creating web applications into the framework itself, taking over mundane tasks such as dispatching incoming requests, constructing and interpretting URLs encoded with information, handling localization and internationalization and much more besides.

The "mantra" of Tapestry is "objects, methods and properties". That is, rather than have developers concerned about the paraphanlia of the Servlet API: requests, responses, sessions, attributes, parameters, URLs and so on, Tapestry focuses the developer on objects (including Tapestry pages and components, but also including the domain objects of the application), methods on those objects, and JavaBeans properties of those objects. That is, in a Tapestry application, the actions of the user (clicking links and submitting forms) results in changes to object properties combined with the invocation of user-supplied methods (containing application logic). Tapestry takes care of the plumbing necessary to connect these user actions with the objects.

This is not to say the Servlet API is inaccessible; it is simply not *relevant* to a typical Tapestry user.

This document describes many of the internals of Tapestry. It is not a tutorial, that is available as a separate document. Instead, this document is a guide to some of the internals of Tapestry, and is intended for experienced developers who wish to leverage Tapestry fully.

Tapestry is currently in release 3.0, and has come a long way in the last couple of years. Tapestry's focus is still on generating dynamic HTML pages.

Nearly all of Tapestry's API is described in terms of interfaces, with default implementations supplied. By substituting new objects with the correct interfaces, the behavior of the framework can be changed significantly. This allows for changes to where Tapestry specifications and templates originate from, and how server-side state is persisted (for example).

Finally, Tapestry boasts extremely complete JavaDoc API documentation. This document exists to supplement that documentation, to fill in gaps that may not be obvious. The JavaDoc is often the best reference.

Pages and Components

Tapestry divides an application into a set of pages. Each page is assembled from Tapestry components. Components themselves may be assembled from other components ... there's no artificial depth limit.

Tapestry pages are themselves components, but are components with some special responsibilities.

All Tapestry components can be containers of other components. Tapestry pages, and most user-defined components, have a template, a special HTML file that defines the static and dynamic portions of the component, with markers to indicate where embedded components are active. Components do not have to have a template, most of the components provided with Tapestry generate their protion of response in code, not using a template.

Components may have one or more named parameters which may be set (or, "bound") by the page or component which contains them. Unlike Java method parameters, Tapestry component parameters may be bidirectional; a component may read a parameter to obtain a value, or write a parameter to set a value.

Most components are concerned only with generating HTML. A certain subset of components deal with the flip-side of requests; handling of incoming requests. Link classes, such as `PageLink`, `DirectLink` and `ActionLink`, create clickable links in the rendered page and are involved in dispatching to user-supplied code when such a link is triggered by clicking it.

Other components, `Form`, and the form element components (`TextField`, `PropertySelection`, `Checkbox`, etc.), facillitate HTML forms. When such components render, they read properties from application objects so as to pro-

vide default values. When forms are submitted, the components within the form read HTTP query parameters, convert the values to appropriate types and then update properties of application objects.

Engines, Services and the Visit

Tapestry has evolved its own jargon over time.

The Engine is a central object, it occupies the same semantic space in Tapestry that the `HttpSession` does in the Servlet API. The Engine is ultimately responsible for storing the persistant state of the application (properties that exist from one request to the next), and this is accomplished by storing the Engine into the `HttpSession`. This document will largely discuss the *default* implementation, with notes about how the default implementation may be extended or overriden, where appropriate.

Engine services are the bridge between servlets and URLs and the rest of Tapestry. Engine services are responsible for encoding URLs, providing query parameters that identify, to the framework, the exact operation that should occur when the generated URL is triggered (by the end user clicking a link or submitting a form). Services are also responsible for dispatching those incoming requests. This encapsulation of URL encoding and decoding inside a single class is key to how Tapestry components can flexibily operate without concern for how they are contained and on which page ... the services take into account page and location when formulating URLs.

The Visit is not a particular object, it is an application-defined object that acts as a focal point for all server-side state (not associated with any single page). Individual applications define for themselves the class of the Visit object. The Visit is stored as a property of the Engine, and so is ultimately stored persistantly in the `HttpSession`

Chapter 2. Tapestry Page and Component Templates

Unlike many other web frameworks, such as Struts or WebWork, Tapestry does not "plug into" an external templating system such as JavaServer Pages or Velocity. Instead, Tapestry integrates its own templating system.

Tapestry templates are designed to look like valid HTML files (component HTML templates will just be snippets). Tapestry "hides" its extensions into special attributes of ordinary HTML elements.

Don't be fooled by the terminology; we say "HTML templates" because that is the prevalent use of Tapestry ... but Tapestry is equally adept at WML or XML.

Template locations

The general rule of thumb is that a page's HTML template is simply an HTML file, stored in the context root directory. That is, you'll have a `MyPage.html` HTML template, a `WEB-INF/MyPage.page` page specification, and a `MyPage` class, in some Java package.

Tapestry always starts knowing the name of the page and the location of the page's specification when it searches for the page's HTML template. Starting with this, it performs the following search:

- In the same location as the specification
- In the web application's context root directory (if the page is an application page, not a page from a library)

In addition, any HTML template in the web application context is considered a page, even if there is no matching page specification. For simple pages that don't need to have any page-specific logic or properties, there's no need for a page specification. Such a page may still use the special Tapestry attributes (described in the following sections).

Finally, with some minor configuration it is possible to change the extension searched for, which is appropriate if you are, for example, building a WML application using Tapestry.

Template Contents

Tapestry templates contain a mix of the following elements:

- Static HTML markup
- Tapestry components
- Localized messages
- Special template directives

Usually, about 90% of a template is ordinary HTML markup. Hidden inside that markup are particular tags that are placeholders for Tapestry components; these tags are recognized by the presence of the `jwcid` attribute. "JWC" is short for "Java Web Component", and was chosen as the "magic" attribute so as not to conflict with any real HTML attribute.

Tapestry's parser is quite flexible, accepting all kinds of invalid HTML markup. That is, attributes don't *have* to be quoted. Start and end tags don't have to balance. Case is ignored when matching start and end tags. Basically, the kind of ugly HTML you'll find "in the field" is accepted.

Components in Templates

Components can be placed anywhere inside a template, simply by adding the `jwcid` attribute to any existing tag. For example:

Example 2.1. Example HTML template containing components

```
<html>
  <head>
    <title>Example HTML Template</title>
  </head>
  <body>
    <span jwcid="border">  ❶

      Hello,
      <span jwcid="@Insert" value="ognl:user.name">Joe User</span> ❷

    </span>
  </body>
</html>
```

❶ This is a reference to a *declared component*; the type and parameters of the component are in the page's specification.

❷ *Implicit component* `Insert` OGNL `user.name`

The point of all this is that the HTML template should preview properly in a WYSIWYG HTML editor. Unlike Velocity or JSPs, there are no strange directives to get in the way of a preview (or necessitate a special editting tool), Tapestry hides what's needed inside existing tags; at worst, it adds a few non-standard attributes (such as `jwcid`) to tags. This rarely causes a problem with most HTML editors.

Templates may contain components using two different styles. *Declared components* are little more than a placeholder; the type of the component is defined elsewhere, in the page (or component) specification.

Alternately, an *implicit component* can be defined in place, by preceding the component type with an "@" symbol. Tapestry includes over forty components with the framework, additional components may be created as part of your application, or may be provided inside a component library.

In the above example, a `<span>` was used for both components. Tapestry doesn't care what tag is used for a component, as long as the start and end tags for components balance (it doesn't even care if the case of the start tag matches the case of the end tag). The example could just as easily use `<div>` or `<fred>`, the rendered page sent back to the client web browser will be the same.

Component Ids

Every component in Tapestry has its own id. In the above example, the first component has the id "border". The second component is anonymous; the framework provides a unique id for the component since one was not supplied in the HTML template. The framework provided id is built from the component's type; this component would have an id of `$Insert`; other `Insert` components would have ids `$Insert$0`, `$Insert$1`, etc.

A component's id must only be unique within its immediate container. Pages are top-level containers, but components can also contain other components.

Implicit components can also have a specific id, by placing the id in front of the "@" symbol:

```
  <span jwcid="insert@Insert" value="ognl:user.name">Joe User</span>
```

The component is still implicit; nothing about the component would go in the specification, but the id of the compo-

nent would be "insert".

Each component may only appear *once* in the template. You simply can't use the same component repeatedly ... but you can duplicate a component fairly easily; make the component a declared component, then use the `copy-of` attribute of the `<component>` element to create clones of the component with new ids.

Specifying Parameters

Using either style, parameters of the component may be bound by adding attributes to the tag. Most attributes bind static parameters, equivalent to using the `<static-binding>` element in the specification.

Prefixing an attribute value with `ognl:` indicates that the value is really an OGNL expression, equivalent to using the `<binding>` element in the specification.

Finally, prefixing an attribute value with `message:` indicates that the value is really a key used to get a localized message, equivalent to the `<message-binding>` element in the specification.

Formal and Informal Parameters

Components may accept two types of parameters: *formal* and *informal*. Formal parameters are those defined in the component specification, using the `<parameter>` element. Informal parameters are *additional* parameters, beyond those known when the component was created.

The majority of components that accept informal parameters simply emit the informal parameters as additional attributes. Why is that useful? Because it allows you to specify common HTML attributes such as `class` or `id`, or JavaScript event handlers, without requiring that each component define each possible HTML attribute (a list which expands all the time).

Informal and formal parameters can be specified in either the specification or in the template. Informal parameters *are not* limited to literal strings, you may use the `ognl:` and `message:` prefixes with them as well.

Not all components allow informal parameters; this is controlled by the `allow-informal-parameters` attribute of the `<component-specification>` element. Many components do not map directly to an HTML element, those are the ones that do not allow informal parameters. If a component forbids informal parameters, then any informal parameters in the specification or the template will result in errors, with one exception: static strings in the HTML template are simply ignored when informal parameters are forbidden; they are presumed to be there only to support WYSIWYG preview.

Each component declares a list of reserved names using the `<reserved-parameter>` element; these are names which are not allowed as informal parameters, because the component generates the named attribute itself, and doesn't want the value it writes to be overriden by an informal parameter. For example, the `DirectLink` component reserves the name "href" ... putting an `href` attribute in the HTML template is simply ignored (again, presumed to be there to support WYSIWYG preview). Case is ignored when comparing attribute names to reserved names.

Chapter 3. Managing Server-Side State

Server-side state is any information that exists on the server, and persists between request cycles. This can be anything from a single flag all the way up to a large database result set. In a typical application, server-side state is the identity of the user (once the user logs in) and, perhaps, a few important domain objects (or, at the very least, primary keys for those objects).

In a typical servlet application, managing server-side state is entirely the application's responsibility. The Servlet API provides just the `HttpSession`, which acts like a `Map`, relating keys to arbitrary objects. It is the application's responsibility to obtain values from the session, and to update values into the session when they change.

Tapestry takes a different tack; it defines server-side state in terms of the Engine, the Visit and persistent page properties.

Understanding Servlet State

Managing server-side state is oneof the most complicated and error-prone aspects of web application design, and one of the areas where Tapestry provides the most benefit. Generally speaking, Tapestry applications which are functional within a single server will be functional within a cluster with no additional effort. This doesn't mean planning for clustering, and testing of clustering, is not necessary; it just means that, when using Tapestry, it is possible to narrow the design and testing focus.

The point of server-side state is to ensure that information about the user acquired during the session is available later in the same session. The canonical example is an application that requires some form of login to access some or all of its content; the identify of the user must be collected at some point (in a login page) and be generally available to other pages.

The other aspect of server-side state concerns failover. Failover is an aspect of highly-available computing where the processing of the application is spread across many servers. A group of servers used in this way is referred to as a *cluster*. Generally speaking (and this may vary significantly between vendor's implementations) requests from a particular client will be routed to the same server within the cluster.

In the event that the particular server in question fails (crashes unexpectedly, or otherwise brought out of service), future requests from the client will be routed to a different, surviving server within the cluster. This failover event should occur in such a way that the client is unaware that anything exceptional has occured with the web application; and this means that any server-side state gathered by the original server must be available to the backup server.

The main mechanism for handling this using the Java Servlet API is the `HttpSession`. The session can store *attributes*, much like a `Map`. Attributes are object values referenced with a string key. In the event of a failover, all such attributes are expected to be available on the new, backup server, to which the client's requests are routed.

Different application servers implement `HttpSession` replication and failover in different ways; the servlet API specification is delibrately unspecific on how this implementation should take place. Tapestry follows the conventions of the most limited interpretation of the servlet specification; it assumes that attribute replication only occurs when the `HttpSession` `setAttribute()` method is invoked [1].

Attribute replication was envisioned as a way to replicate simple, immutable objects such as `String` or `Integer`. Attempting to store mutable objects, such as `List`, `Map` or some user-defined class, can be problematic. For example, modifying an attribute value after it has been stored into the `HttpSession` may cause a failover error. Effectively, the backup server sees a snapshot of the object at the time that `setAttribute()` was invoked; any later change to the object's internal state is *not* replicated to the other servers in the cluster! This can result in strange and unpredictable behavior following a failover.

Tapestry attempts to sort out the issues involving server-side state in such a way that they are invisible to the developer. Most applications will not need to explicitly access the `HttpSession` at all, but may still have significant amounts of server-side state. The following sections go into more detail about how Tapestry approaches these is-

1 This is the replication strategy employed by BEA's WebLogic server.

sues.

Engine

The engine, a class which implements the interface `IEngine`, is the central object that is responsible for managing server-side state (among its many other responsibilities). The engine is itself stored as an `HttpSession` attribute.

Because the internal state of the engine can change, the framework will re-store the engine into the `HttpSession` at the end of most requests. This ensures that any changes to the Visit object are properly replicated.

The simplest way to replicate server-side state is simply not to have any. With some care, Tapestry applications can run stateless, at least until some actual server-side state is necessary.

Visit Object

The Visit object is an application-defined object that may be obtained from the engine (via the `visit` property of the `IEngine` or `IPage`). By convention, the class is usually named `Visit`, but it can be any class whatsoever, even `Map`.

The name, "Visit", was selected to emphasize that whatever data is stored in the Visit concerns just a single visit to the web application. Tapestry is stricly concerned with providing the presentation layer of the application; it doesn't include any kind of database access, or any other kind of long-term data storage. However, it is very easy to interface a Tapestry application to any kind of backend system.

The following example demonstrates how a listener method may access the visit object.

Example 3.1. Accessing the Visit object

```
public void formSubmit(IRequestCycle cycle)
{
    Visit visit = (Visit)getPage().getVisit();

    visit.doSomething();
}
```

In most cases, listener methods, such as `formSubmit()`, are implemented directly within the page. In that case, the first line can be abbreviated to:

```
    Visit visit = (Visit)getVisit();
```

The Visit object is instantiated lazily, the first time it is needed. Method `createVisit()` of `AbstractEngine` is responsible for this.

In most cases, the Visit object is an ordinary JavaBean, and therefore, has a no-arguments constructor. In this case, the complete class name of the Visit is specified as configuration property `org.apache.tapestry.visit-class`.

Typically, the Visit class is defined in the application specification, or as a `<init-parameter>` in the web deployment descriptor (web.xml).

Example 3.2. Defining the Visit class

```
```

```
<application name="Tapestry Component Workbench">
  <property name="org.apache.tapestry.visit-class" value="org.apache.tapestry.workbench.Visit"/>

  ...
```

In cases where the Visit object does not have a no-arguments contructor, or has other special initialization requirements, the method `createVisit()` of `AbstractEngine` can be overridden.

There is a crucial difference between accessing the visit via the `visit` property of `IPage` and the `visit` property of `IEngine`. In the former case, accessing the visit via the page, the visit *will* be created if it does not already exist.

Accessing the visit via the `IEngine` is different, the visit will *not* be created if it does not already exist.

Carefully crafted applications will take heed of this difference and try to avoid creating the visit unnecessarilly. It is not just the creation of this one object that is to be avoided ... creating the visit will likely force the entire application to go stateful (create an `HttpSession`), and applications are more efficient while stateless.

Global Object

The application Global object is very similar to the application Visit object with some key differences. The Global object is shared by all instances of the application engine; ultimately, it is stored as a `ServletContext` attribute. The Global object is therefore not persistent in any way. In a failover, the engine will connect to a new instance of the Global object within the new server.

The Global object may be accessing using the `global` property of either the page or the engine (unlike the `visit` property, they are completely equivalent).

Care should be taken that the Global object is threadsafe; since many engines (from many sessions, in many threads) will access it simultanenously. The default Global object is a synchronized `HashMap`. This can be overriden with configuration property `org.apache.tapestry.global-class`.

The most typical use of the Global object is to interface to J2EE resources such as EJB home and remote interfaces or JDBC data sources. The shared Global object can cache home and remote interfaces that are efficiently shared by all engine instances.

Persistent Page Properties

Servlets, and by extension, JavaServer Pages, are inherently stateless. That is, they will be used simultaneously by many threads and clients. Because of this, they must not store (in instance variables) any properties or values that are specified to any single client.

This creates a frustration for developers, because ordinary programming techniques must be avoided. Instead, client-specific state and data must be stored in the `HttpSession` or as `HttpServletRequest` attributes. This is an awkward and limiting way to handle both *transient* state (state that is only needed during the actual processing of the request) and *persistent* state (state that should be available during the processing of this and subsequent requests).

Tapestry bypasses most of these issues by *not* sharing objects between threads and clients. For the duration of a request, a page and all components within the page are reserved to the single request. There is no chance of conflicts because only the single thread processing the request will have access to the page. At the end of the request cycle, the page is reset back to a pristine state and returned to the shared pool, ready for reuse by the same client, or by a different client.

In fact, even in a high-volume Tapestry application, there will rarely be more than a few instances of any particular page in the page pool.

For this scheme to work it is important that at the end of the request cycle, the page must return to its pristine state. The prisitine state is equivalent to a freshly created instance of the page. In other words, any properties of the page

that changed during the processing of the request must be returned to thier initial values.

Tapestry separates the persistent state of a page from any instance of the page. This is very important, because from one request cycle to another, a different instance of the page may be used ... even when clustering is not used. Tapestry has many copies of any page in a pool, and pulls an arbitrary instance out of the pool for each request.

In Tapestry, a page may have many properties and may have many components, each with many properties, but only a tiny number of all those properties needs to persist between request cycles. On a later request, the same or different page instance may be used. With a little assistance from the developer, the Tapestry framework can create the illusion that the same page instance is being used in a later request.

Each persistent page property is stored individually as an `HttpSession` attribute. Like the Servlet API, persistent properties work best with immutable objects such as `String` and Integer;. For mutable objects (including `List` and `Map`), Tapestry makes a *copy* of the property. In the worst case, Tapestry may have to serialize and deserialize the object to make a copy. Using several properties with simple, immutable types is therefore much less expensive than using a single, custom, complex, mutable object.

Persistent properties make use of a `<property-specification>` element in the page or component specification. Tapestry does something special when a component contains any such elements; it dynamically generates a subclass that provides the desired fields, methods and whatever extra initialization or cleanup is required.

You may also, optionally, make your class abstract, and define abstract accessor methods that will be filled in by Tapestry in the generated subclass. This allows you to read and update properties inside your class, inside listener methods.

Note

Properties defined this way may be either transient or persistent. It is useful to define even transient properties using the `<property-specification>` element because doing so ensures that the property will be properly reset at the end of the request (before the page is returned to the pool for later reuse).

Example 3.3. Persistent Page Property: Java Class

```
public abstract class MyPage extends BasePage
{
    abstract public int getItemsPerPage();

    abstract public void setItemsPerPage(int itemsPerPage);
}
```

Example 3.4. Persistent Page Property: Page Specification

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification PUBLIC
  "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
  "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification class="MyPage">

  <property-specification name="itemsPerPage" persistent="yes" type="int" initial-value="10"/>

</page-specification>
```

Again, making the class abstract, and defining abstract accessors is *optional*. It is only useful when a method within the class will need to read or update the property. It is also valid to just implement one of the two accessors. The enhanced subclass will always include both a read and a write accessor.

This exact same technique can be used with components as well as pages.

A last note about initialization. After Tapestry invokes the `finishLoad()` method, it processes the initial value provided in the specification. If the `initial-value` attribute is ommitted or blank, no change takes place. Tapestry then takes a snapshot of the property value, which it retains and uses at the end of each request cycle to reset the property back to its "pristine" state.

This means that you may perform initialization for the property inside `finishLoad()` (instead of providing a `initial-value`). However, don't attempt to update the property from `initialize()` ... the order of operations when the page detaches is not defined and is subject to change.

Implementing Persistent Page Properties Manually

Warning

There is very little reason to implement persistent page properties manually. Using the `property-<specification>` element is much easier, and nearly as efficient. It is highly unlikely that the extra developer effort used to implement persistent page properties manually will pay off in any improvement in application throughput.

The preferred way to implement persistent page properties without using the `<property-specification>` element is to implement the method `initialize()` on your page. This method is invoked once when the page is first created; it is invoked again at the end of each request cycle. An empty implementation of this method is provided by `AbstractPage`.

The first example demonstrates how to properly implement a transient property. It is simply a normal JavaBean property implementation, with a little extra to reset the property back to its pristine value (`null`) at the end of the request.

Example 3.5. Use of `initialize()` method

```
public class MyPage extends BasePage
{
    private String _message;

    public String getMessage()
    {
        return _message;
    }

    public void setMessage(String message)
    {
        _message = message;
    }

    protected void initialize()
    {
        _message = null;
    }
}
```

If your page has additional attributes, they should also be reset inside the `initialize()` method.

Now that we've shown how to manually implement *transient* state, we'll show how to handle *persistent* state.

For a property to be persistent, all that's necessary is that the accessor method notify the framework of changes. Tapestry will record the changes (using an `IPageRecorder`) and, in later request cycles, will restore the property using using the recorded value and whichever page instance is taken out of the page pool.

This notification takes the form of an invocation of the static method `fireObservedChange()` in the `Tapestry` class. This method is overloaded for all the scalar types, and for `Object`.

Example 3.6. Manual Persistent Page Property

```
public class MyPage extends BasePage
{
    private int _itemsPerPage;

    public int getItemsPerPage()
    {
        return _itemsPerPage;
    }

    public void setItemsPerPage(int itemsPerPage)
    {
        _itemsPerPage = itemsPerPage;

        Tapestry.fireObservedChange(this, "itemsPerPage", itemsPerPage);
    }

    protected void initialize()
    {
        _itemsPerPage = 10;
    }
}
```

This sets up a property, `itemsPerPage`, with a default value of 10. If the value is changed (perhaps by a form or a listener method), the changed value will "stick" with the user who changed it, for the duration of their session.

Manual Persistent Component Properties

Implementing transient and persistent properties inside components involves more work. The `fireObservedChange()` method is available to components as well as pages, but the initialization of the component is more complicated.

Components do not have the equivalent of the `initialize()` method. Instead, they must register for an event notification to tell them when the page is being *detached* from the engine (prior to be stored back into the page pool). This event is generated by the page itself.

The Java interface `PageDetachListener` is the event listener interface for this purpose. By simply implementing this interface, Tapestry will register the component as a listener and ensure that it receives event notifications at the right time (this works for the other page event interface, `PageRenderListener` as well; simply implement the interface and leave the rest to the framework).

Tapestry provides a method, `finishLoad()`, for just this purpose: late initialization.

Example 3.7. Manual Persistent Component Properties

```
public class MyComponent extends BaseComponent implements PageDetachListener
{
    private String _myProperty;

    public void setMyProperty(String myProperty)
    {
        _myProperty = myProperty;

        Tapestry.fireObservedChange(this, "myProperty", myProperty);
    }

    public String getMyProperty()
    {
        return _myProperty;
    }

    protected void initialize()
    {
        _myProperty = "a default value";
    }

    protected void finishLoad()
    {
        initialize();
    }

    /**
     *  The method specified by PageDetachListener.
     *
     **/

    public void pageDetached(PageEvent event)
    {
        initialize();
    }
}
```

Again, there is no particular need to do all this; using the `<property-specification>` element is far, far simpler.

Stateless Applications

In a Tapestry application, the framework acts as a buffer between the application code and the Servlet API ... in particular, it manages how data is stored into the `HttpSession`. In fact, the framework controls *when* the session is first created.

This is important and powerful, because an application that runs, even just initially, without a session consumes less resources that a stateful application. This is even more important in a clustered environment with multiple servers; any data stored into the `HttpSession` will have to be replicated to other servers in the cluster, which can be expensive in terms of resources. Using less resources means better throughput and more concurrent clients, always a good thing in a web application.

Tapestry defers creation of the `HttpSession` until one of two things happens: When the visit is created, or when the first persistent page property is recorded. At this point, Tapestry will create the `HttpSession` and store the engine into it.

Earlier, we said that the `IEngine` instance is stored in the `HttpSession`, but this is not always the case. Tapestry maintains a pool of `IEngine` instances that are used for stateless requests. An instance is checked out of the pool and used to process a single request, then checked back into the pool for reuse in a later request, by the same or different client.

For the most part, your application will be unaware of when it is stateful or stateless; statefulness just happens on its own. Ideally, at least the first, or "Home" page, should be stateless (it should be organized in such a way that the visit is not created, and no persistent state is stored). This will help speed the initial display of the application, since

no processing time will be used in creating the session.

Chapter 4. Configuring Tapestry

Web Deployment Descriptor

All Tapestry applications make use of the `ApplicationServlet` class as their servlet; it is rarely necessary to create a subclass.

Example 4.1. Virtual Library 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>
  <distributable/> ❶
  <display-name>Tapestry Virtual Library Demo</display-name>
  <servlet>
    <servlet-name>vlib</servlet-name> ❷
    <servlet-class>org.apache.tapestry.ApplicationServlet</servlet-class> ❸
    <init-param>
            <param-name>org.apache.tapestry.application-specification</param-name> ❹
            <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> ❺
  </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>
```

❶ This indicates to the application server that the Tapestry application may be clustered. Most application servers ignore this element, but future servers may only distribute applications within a cluster if this element is present.

JBoss is very literal!

JBoss 3.0.x appears to be very literal about the `<distributable>` element. If it appears, you had better be deploying into a clustered environment, otherwise HttpSession state management simply doesn't work.

❷ The servlet name may be used when locating the application specification (though not in this example).

❸ The servlet class is nearly always `ApplicationServlet`. There's rarely a need to create a subclass; Tapestry has many other hooks for extending the application.

❹ The Virtual Library application stores its specification on the classpath, rather than under `WEB-INF`, so it is necessary to provide the complete path to the specification. Most applications can omit this `<init-param>`.

❺ The servlet is mapped to `/app` within the context. The context itself has a path, determined by the application server, but typically `/vlib`. The client web browser will see the Tapestry application as `/` `http://`*`host`*`vlib/app`.

Using `/app` as the URL is a common convention when creating Tapestry applications, but is not a requirement.

On initialization, the Tapestry servlet will locate its application specification; a file that identifies details about the application, the pages and components within it, and any component libraries it uses. Tapestry provides a great deal of flexibility on where the specification is stored; trivial Tapestry applications can operate without an application specification.

Prior to release 3.0, application specifications had to be stored on the classpath. This is maintained for backwards compatibility. In modern applications, the specification is stored under `WEB-INF`. In fact, Tapestry performs a search to find the specification:

1. On the classpath, as defined by the `org.apache.tapestry.application-specification` configuration parameter.
2. As `/WEB-INF/`*`name`*`/`*`name`*`.application`. The *`name`* is the servlet name. This location is only used in the rare case of a single WAR containing multiple Tapestry applications.
3. As `/WEB-INF/`*`name`*`.application`. Again, *`name`* is the servlet name. This is the standard scenario.

If the application specification still can not be found, then an empty, "stand in" application specification is used.

Configuration Search Path

Tapestry occasionally must obtain a value for a configuration property. These configuration properties are items that are frequently optional, and don't fit into any particular specification. Many are related to the runtime environment, such as which class to instantiate as the Visit object.

Tapestry is very flexible about where values for such properties may be obtained. In general, the search path for configuration properties is:

- As a `<property>` of the `<application>` (in the application specification, if the application uses one).
- As an `<init-parameter>` for the servlet, in the web application deployment descriptor.
- As an `<init-parameter>` for the servlet context, also in the web application deployment descriptor.
- As a JVM system property.
- Hard-coded "factory" defaults (for some properties).

It is expected that some configurations are not defined at any level; those will return null.

Applications are free to leverage this lookup mechanism as well. `IEngine` defines a `propertySource` property (of type `IPropertySource`) that can be used to perform such lookups.

Applications may also want to change or augment the default search path; this is accomplished by overriding `AbstractEngine` method `createPropertySource()`. For example, some configuration data could be drawn from a database.

The following table lists all the configuration values currently used in Tapestry.

Configuration Values

org.apache.tapestry.template-extension
: Overrides the default extension used to locate templates for pages or components. The default extension is "html", this configuration property allows overrides where appropriate. For example, an application that produces WML may want to override this to "wml".

  This configuration property does not follow the normal search path rules. The `<property>` must be provided in the `<page-specification>` or `<component-specification>`. If no value is found there, the immediate containing `<application>` or `<library-specification>` is checked. If still not found, the default is used.

org.apache.tapestry.asset.dir, org.apache.tapestry.asset.URL
: These two values are used to handle private assets. Private assets are assets that are stored on the classpath, and not normally visible to client web browsers.

  By specifying these two configuration values, Tapestry can export private assets to a directory that is visible to the client web browser. The `URL` value should map to the directory specified by the `dir` value.

org.apache.tapestry.visit-class
: The fully qualified class name to instantiate as the Visit object.

  If not specified, an instance of `HashMap` will be created.

org.apache.tapestry.default-page-class
: By default, any page that omits the `class` attribute (in its `<page-specification>`) will be instantiated as `BasePage`. If this is not desired, the default may be overridden by specifying a fully qualified class name.

org.apache.tapestry.engine-class
: The fully qualified class name to instantiate as the application engine. This configuration value is only used when the application specification does not exist, or fails to specify a class. By default, `BaseEngine` is used if this configuration value is also left unspecified.

org.apache.tapestry.global-class
: The fully qualified class name to instantiate as the engine `global` property. The Global object is much like Visit object, except that it is shared by all instances of the application engine rather than being private to any particular session. If not specified, a synchronized instance of `HashMap` is used.

org.apache.tapestry.default-script-language
: The name of a BSF-supported language, used when a `<listener-binding>` element does not specify a language. If not overridden, the default is "jython".

org.apache.tapestry.enable-reset-service
: If not specified as "true", then the `reset` service will be non-functional. The reset service is used to force the running Tapestry application to discard all cached data (including templates, specifications, pooled objects and more). This must be explicitly enabled, and should only be used in development (in production, it is too easily exploited as a denial of service attack).

  Unlike most other configuration values, this must be specified as a JVM system property.

org.apache.tapestry.disable-caching
: If specified (as "true"), then the framework will discard all cached data (specifications, templates, pooled objects, etc.) at the end of each request cycle.

  This slows down request handling by a noticable amount, but is very useful in devlopment; it means that changes to templates and specifications are immediately visible to the application. It also helps identify any errors in managing persistent page state.

  This should never be enabled in production; the performance hit is too large. Like `org.apache.tapestry.enable-reset-service`, this must be specified as a JVM system property.

Application Extensions

Tapestry is designed for flexibility; this extends beyond simply configuration behavior, and encompasses actually replacing or augmenting behavior. In some cases, it is necessary to subclass framework classes in order to alter behavior, but in many cases, it is possible to use an application extension.

Application extensions are JavaBeans declared in the application specification using the `<extension>` element. Each extension consists of a name, a Java class to instantiate, and an optional configuration (that is, properties of the bean may be set). The framework has a finite number of extension points. If an extension bean with the correct name exists, it will be used at that extension point.

Each application extension must implement an interface particular to the extension point.

Application Extension Points

org.apache.tapestry.property-source (`IPropertySource`)
: This extension is fit into the configuration property search path, after the servlet context, but before JVM system properties. A typical use would be to access some set of configuration properties stored in a database.

org.apache.tapestry.request-decoder (`IRequestDecoder`)
: A request decoder is used to identify the actual server name, server port, scheme and request URI for the request. In some configurations, a firewall may invalidate the values provided by the actual `HttpServletRequest` (the values reflect the internal server forwarded to by the firewall, not the actual values used by the external client). A request decoder knows how to determine the actual values.

org.apache.tapestry.monitor-factory (`IMonitorFactory`)
: An object that is used to create `IMonitor` instances. Monitors are informed about key application events (such as loading a page) during the processing of a request.

  The factory may create a new instance for the request, or may simply provide access to a shared instance.

  If not specified, a default implementation is used (`DefaultMonitorFactory`).

org.apache.tapestry.specification-resolver-delegate (`ISpecificationResolverDelegate`)
: An object which is used to find page and component specifications that are not located using the default search rules. The use of this is open-ended, but is generally useful in very advanced scenarios where specifications are stored externally (perhaps in a database), or constructed on the fly.

org.apache.tapestry.template-source-delegate (`ITemplateSourceDelegate`)
: An object which is used to find page or component templates that are not located using the default search rules. The use of this is open-ended, but is generally useful in very advanced scenarios where templates are stored externally (perhaps in a database), or constructed on the fly.

org.apache.tapestry.multipart-decoder (`IMultipartDecoder`)
: Allows an alternate object to be responsible for decoding multipart requests (context type multipart/form-data, used for file uploads). Generally, this is used to configure an instance of `DefaultMultipartDecoder` with non-default values for the maximum upload size, threshold size (number of bytes before a temporary file is created to store the) and repository directory (where temporary files are stored).

Appendix A. Tapestry Specification DTDs

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

Table A.1. Tapestry Specifications

Type	File Extension	Root Element	Public ID	System ID
Application	application	`<application>`	`-//Apache Software Foundation//Tapestry Specification 3.0//EN`	`http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd`
Page	page	`page-<specification>`	`-//Apache Software Foundation//Tapestry Specification 3.0//EN`	`http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd`
Component	jwc	`component-<specification>`	`-//Apache Software Foundation//Tapestry Specification 3.0//EN`	`http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd`
Library	library	`library-<specification>`	`-//Apache Software Foundation//Tapestry Specification 3.0//EN`	`http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd`
Script	script	`<script>`	`-//Apache Software Foundation//Tapestry Script Specification 3.0//EN`	`http://jakarta.apache.org/tapestry/dtd/Script_3_0.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 A.1. `<application>` Attributes

Name	Type	Required ?	Default Value	Description
name	string	no		User presentable name of application.
engine-class	string	no		Name of an implementation of

Name	Type	Required ?	Default Value	Description
				`IEngine` to instantiate. Defaults to `BaseEngine` if not specified.

Figure A.2. <application> Elements

```
<description> ?, <property> *,
(<page> | <component-type> | <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-message-property> elements.

Figure A.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|render|request`	no	`request`	As described above; duration that bean is retained.

Figure A.4. <bean> Elements

```
<description> ?, <property> *,
(<set-property> | <set-message-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`*.

If the `expression` attribute is omitted, then the body of the element is used. This is useful when the expression is long, or contains problematic characters (such as a mix of single and double quotes).

Figure A.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.

<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 A.6. <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.
inherit-informal-parameters	yes|no	no	no	If yes, then any informal parameters of the containing component will be copied into 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 A.7. <component> Elements

```
<property> *,
(<binding> | <inherited-binding> | <listener-binding> | <static-binding> | <message-binding>) *
```

<component-type> 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 A.8. <component-type> Attributes

Name	Type	Required ?	Default Value	Description
type	string	yes		A name to be used as a component type.
specification-path	string	yes		An absolute or relative resource path to the component's specification (including leading slash and file extension). Relative resources are evaluated relative to the location of the containing application or library specfication.

`<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 A.9. `<component-specification>` Attributes

Name	Type	Required ?	Default Value	Description
class	string	no		The Java class to instantiate, which must implement the interface `IComponent`. If not specified, `BaseComponent` is used.
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 `RenderBody` component. If `no`, then any body for this component is discarded.
allow-informal-parameters	`yes|no`	no	`yes`	If `yes`, then any informal parameters (bindings that don't match a formal pa-

Name	Type	Required ?	Default Value	Description
				rameter) specified here, or in the component's tag within its container's template, are retained. Typically, they are converted into additional HTML attributes. If no, then informal parameters are not allowed in the specification, and discarded if in the template.

Figure A.10. <component-specification> Elements

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

<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 A.11. <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.
value		no		The value to configure, which will be converted before being assigned to the property. If not provided, the character data wrapped by the element is used instead.

<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`*.

The path may be either absolute or relative. Absolute paths start with a leading slash, and are evalulated relative to the context root. Relative paths are evaluated relative to the application root, which is typically the same as the context root (the exception being a WAR that contains multiple Tapestry applications, within multiple subfolders).

Figure A.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.

<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.

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.

The <description> element has no attributes.

<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 A.13. <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 construc-

Name	Type	Required ?	Default Value	Description
				tor.
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 A.14. <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 A.15. <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 access the asset.

<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 A.16. <inherited-binding> Attributes

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

Name	Type	Required ?	Default Value	Description
parameter-name	string	yes		The name of a parameter 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 A.17. `<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 A.18. `<library-specification>` Elements

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

`<listener-binding>` element

Appears in: `<component>`

A listener binding is used to create application logic, in the form of a listener (for a `DirectLink`, `ActionLink`, `Form`, etc.) in place within the specification, in a scripting language (such as Jython or JavaScript). The script itself is the wrapped character data for the `<listener-binding>` element.

When the listener is triggered, the script is executed. Three beans, `page`, `component` and `cycle` are pre-declared.

The `page` is the page activated by the request. Usually, this is the same as the page which contains the `component` ... in fact, usually `page` and `compoment` are identical.

The `component` is the component from whose specification the binding was created (that is, not the `DirectLink`, but the page or component which embeds the `DirectLink`).

The `cycle` is the active request cycle, from which service parameters may be obtained.

Figure A.19. `<listener-binding>` Attributes

Name	Type	Required ?	Default Value	Description
name	string	yes		The name of the listener parameter to bind.
language	string	no		The name of a BSFsupported language that the script is written in. The default, if not specified, is `jython`.

`<message-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 A.20. `<message-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.

`<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 A.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 path to the page's specification, which may be absolute (start with a leading slash), or relative to the application or library 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 A.22. <page-specification> Attributes

Name	Type	Required ?	Default Value	Description
class	string	no		The Java class to instantiate, which must implement the interface IPage. Typically, this is BasePage or a subclass of it. BasePage is the default if not otherwise specified.

Figure A.23. <page-specification> Elements

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

<parameter> element

Appears in: `<component-specification>`

Defines a formal parameter of a component. Parameters may be connected (`in`, `form` or `auto`) or unconnected (`custom`). If a parameter is connected, but the class does not provide the property (or does, but the accessors are abstract), then the framework will create and use a subclass that contains the implementation of the necessary property.

For `auto` parameters, the framework will create a synthetic property as a wrapper around the binding. Reading the property will read the value from the binding and updating the property will update the binding value. `auto` may only be used with required parameters. `auto` is less efficient than `in`, but can be used even when the component is not rendering.

Figure A.24. `<parameter>` Attributes

Name	Type	Required ?	Default Value	Description
name	string	yes		The name of the parameter, which must be a valid Java identifier.
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|auto|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

Name	Type	Required ?	Default Value	Description
				through the binding. `in` means the property is set from the parameter before the component renders, and is reset back to default value after the component renders. `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. `auto` creates a synthetic property that works with the binding to read and update. `auto` parameters must be required, but can be used even when the component is not rendering.
default-value	OGNL expression	no		Specifies the default value for the parameter, if the parameter is not bound.

`<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`*.

The resource path may either be complete and absolute, and start with a leading slash, or be relative. Relative paths are evaluated relative to the location of the containing specification.

Figure A.25. `<private-asset>` Attributes

Name	Type	Required ?	Default Value	Description
name	string	yes		The name of the asset, which must be a

Name	Type	Required ?	Default Value	Description
				valid Java identifier.
resource-path	string	yes		The absolute or relative path to the asset on the classpath.

<property> element

Appears in: *many*

The <property> element is used to store meta-data about some other element (it is contained within). Tapestry ignores this meta-data Any number of name/value pairs may be stored. The value is provided with the value attribute, or the character data for the <property> element.

Figure A.26. <property> Attributes

Name	Type	Required ?	Default Value	Description
name	string	yes		The name of the property.
value	string	no		The value for the property. If omitted, the value is taken from the character data (the text the tag wraps around). If specified, the character data is ignored.

<property-specification> element

Appears in: <component-specification>, <page-specification>

Defines a transient or persistent property to be added to the page or component. Tapestry will create a subclass of the page or component class (at runtime) and add the necessary fields and accessor methods, as well as end-of-request cleanup.

It is acceptible for a page (or component) to be abstract, and have abstract accessor methods matching the names that Tapestry will generate for the subclass. This can be useful when setting properties of the page (or component) from a listener method.

A connected parameter specified in a <parameter> element may also cause an enhanced subclass to be created.

An initial value may be specified as either the initial-value attribute, or as the body of the property-<specification> element itself.

Figure A.27. <property-specification> Attributes

Name	Type	Required ?	Default Value	Description
name	string	yes		The name of the property to create.
type	string	no	java.lang.Object	The type of the property. If abstract accessors exist, they must exactly match this type. The type may be either a fully qualified class name, or the name of one of the basic scalar types.
persistent	`yes|no`	no	`no`	If true, the generated property will be persistent, firing change notifications when it is updated.
initial-value	string	no		An optional OGNL expression used to initialize the property. The expression is evaluated only when the page is first constructed.

`<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 A.28. `<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 A.29. `<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-message-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 A.30. `<set-message-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.

`<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).

The value to be assigned to the bean property can be specified using the `expression` attribute, or as the content of the `<set-property>` element itself.

Figure A.31. `<set-property>` Attributes

Name	Type	Required ?	Default Value	Description
name	string	yes		The name of the helper bean property to set.
expression	string	no		The OGNL expression used to set the property.

`<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 specified as the `value` attribute, or as 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 A.32. `<static-binding>` Attributes

Name	Type	Required ?	Default Value	Description
name	string	yes		The name of the parameter to bind.
value	string	no		The string value to be used. If omitted, the wrapped character data is used instead (which is more convienient if the value is large, or contains problematic punctuation).

Appendix B. 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 B.1. <body> Elements

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

<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 B.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.
index	string	no		If specified, then the named symbol is updated with each successive index.

Figure B.3. <foreach> Elements

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

<if> element

Appears in: *many*

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

Figure B.4. <if> Attributes

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

Figure B.5. <if> Elements

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

<if-not> element

Appears in: *many*

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

Figure B.6. <if-not> Attributes

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

Figure B.7. <if-not> Elements

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

`<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 B.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 B.9. `<initialization>` Elements

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

`<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 B.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 complete, qualified class name for the symbol. The provided symbol must be assignable to this

Name	Type	Required ?	Default Value	Description
				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 B.11. `<let>` Attributes

Name	Type	Required ?	Default Value	Description
key	string	yes		The key of the symbol to define.
unique	boolean	`yes|no`	no	If yes, then the string is ensured to be unique (by possibly adding a suffix) before being assigned to the symbol.

Figure B.12. `<let>` Elements

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

`<script>` element

Root element

The root element of a Tapestry script specification.

Figure B.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 B.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.

<unique> element

Appears in: *many*

Creates a block whose contents are contributed only once, no matter how many times the block is evaluated during the rendering of a single page.

Figure B.15. <unique> Elements

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