Tapestry User's Guide

Howard Lewis Ship

Tapestry User's Guide

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 can take some getting used to. You don't write servlets in Tapestry, you write *listener methods*. You don't build URLs to servlets either -- you use an existing component (such as `DirectLink`) and configure its `listener` parameter to invoke your listener method. What does a listener method do? It interacts with backend systems (often, stateless session EJBs) or does other bookkeeping related to the request and selects a new page to provide a response to the user ... basically, the core code at the center of a servlet. In Tapestry, you write much less code because all the boring, mechanical plumbing (creating URLs, dispatching incoming requests, managing server-side state, and so forth) is the responsibility of the framework.

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 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, although there's plenty of support for XHTML, WML and other types of markup as well.

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. A common example is to override where page and component specifications are stored (perhaps in a database).

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.

An overview of Tapestry

Perhaps the hardest part of understanding Tapestry is the fact that it is *component-centric* not *operation-centric*. Most web technologies (Struts, servlets, PHP, etc.) are operation-centric. You create servlets (or `Action`s, or what have you) that are invoked when a user clicks a link or submits a form. You are responsible for selecting an appropriate URL, and the name and type of any query parameters, so that you can pass along the information you need in the URL.

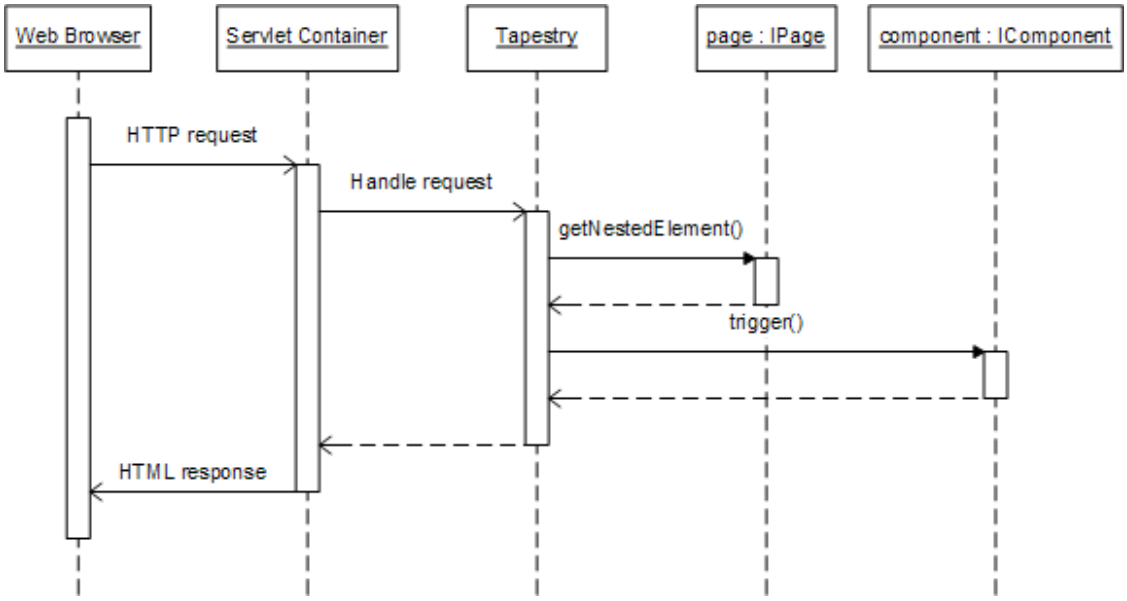
You are also responsible for connecting your output pages (whether they are JSPs, Velocity templates,

or some other form of templating technology) to those operations. This requires you to construct those URLs and get them into the `href` attribute of your `<a>` tag, or into the `action` attribute of your `<form>` tag.

Everything is different inside Tapestry. Tapestry applications consist of pages; pages are constructed from smaller components. Components may themselves be constructed from other components. Every page has a unique name, and every component within a page has its own unique id ... this is a *component object model*. Effectively, every component has an *address* that can easily be incorporated into a URL.

In practical terms, your don't write a servlet for the `add-item-to-shopping-cart` operation. In fact, you don't even write an `add-item-to-shopping-cart` component. What you do is take an existing component, such as `DirectLink`, and configure it. When the component renders, it will create a callback URL. When you click that link, the callback URL (which includes the name of the page and the id of the component within the page) will invoke a method on the component ... and *that* method invokes your application-specific *listener method*. [1] You supply just the listener method ... not an entire servlet. Tapestry takes care that your listener method is invoked at the right time, under the right conditions. You don't have to think about how to build that URL, what data goes in the URL, or how to hook it up to your application-specific code--that's all handled by the framework.

Figure 1.1. Tapestry request dispatch (high level)



Tapestry uses a component object model to dispatch incoming requests to the correct page and component.

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.

[1] Listener methods in Tapestry are very similar in intent to *delegates* in C#. In both cases, a method of a particular object instance is represented as an object. Calling this a "listener" or a "listener method" is a bit of a naming snafu; it should be called a "delegate" and a "delegate method" but the existing naming is too deeply entrenched to change any time soon.

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 portion of response in code, not using a template.

Components may have one or more named parameters which may be set (or, more correctly, "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 control components (`TextField`, `PropertySelection`, `Checkbox`, etc.), facilitate HTML forms. When such components render, they read properties from application objects so as to provide default values. When the 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 friends

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 persistent 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 object 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 object 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 persistently in the `HttpSession`

The Global object is also application-specific. It stores information global to the entire application, independent of any particular user or session. A common use for the Global object is to centralize logic that performs JNDI lookups of session EJBs.

Object Graph Navigation Language

Tapestry is tightly integrated with OGNL, the Object Graph Navigation Language. OGNL is a Java expression language, which is used to peek into objects and read or update their properties. OGNL is similar to, and must more powerful than, the expression language built into the JSP 2.0 standard tag library. OGNL not only support property access, it can include mathematical expressions and method invocations. It can reference static fields of public classes. It can create new objects, including lists and maps.

The simplest OGNL expressions are property names, such as `foo`, which is equivalent to method `getFoo()` (or `setFoo()` if the expression is being used to update a property). The "Navigation" part comes into play when the expression is a series of property names, such as `foo.bar.baz`, which is equivalent to `getFoo().getBar().getBaz()` ... though care must always be taken that the intermediate properties (`foo` and `bar` in this example) are not null.

OGNL is primarily used to allow two different objects (such as a page and a component contained by that page) to share information.

Chapter 2. 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 of HTML rather than complete pages). 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 component 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 used for templates. For example, if you are developing a WML application, you may wish to name your files with the extension `.wml`.

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.

The goal is to allow you to preview your HTML templates using a WYSIWYG HTML editor (or even an ordinary web browser). The editor will ignore the undefined HTML attributes (such as `jwcid`).

A larger goal is to support real project teams: The special markup for Tapestry is unobtrusive, even invisible. This allows an HTML designer to work on a template without breaking the dynamic portions of it. This is completely unlike JSPs, where the changes to support dynamic output are extremelyobtrusive and result in a file that is meaningless to an HTML editor.

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 This is a *implicit component*; the type of the component is `Insert`. The value parameter is bound to the OGNL expression `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.

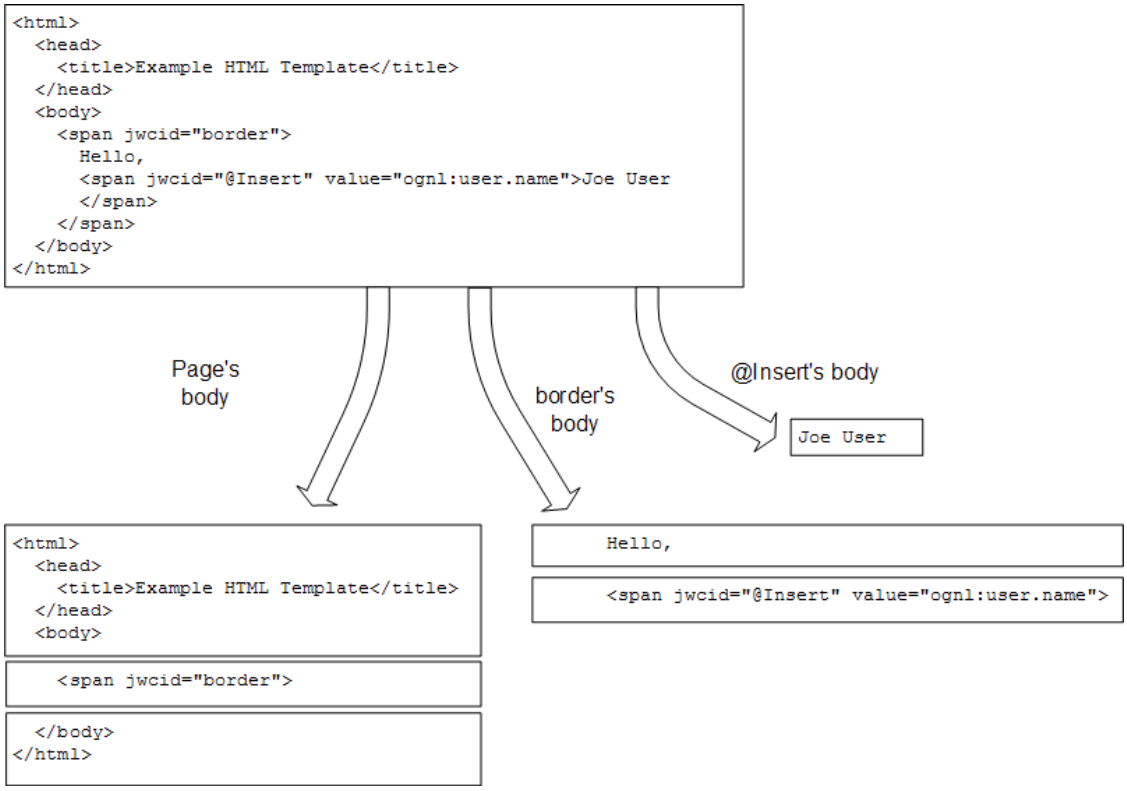
Alternately, an *implicit component* can be defined in place, by preceding the component type with an "@" symbol. Tapestry includes over forty components with the framework, additional components may be created as part of your application, or may be provided inside a component library.

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 bodies

In Tapestry, each component is responsible for rendering itself and its *body*. A component's body is the portion of its page's template [2] that its tags encloses. The Tapestry HTML template parser is responsible for dividing up the template into chunks: blocks of static HTML, component start tags (recognized by the `jwcid` attribute) and matching component end tags. It is quite forgiving about case, quotes (which may be single quotes, double quotes, or even omitted), and missing close tags (except for components, which must be balanced).

Figure 2.1. Component templates and bodies



The template is broken into small chunks that are each slotted into a particular component's body.

In most cases, a component will make use of its body; it simply controls if, when and how often its body is rendered (when rendering the HTML response sent to the client). Other components, such as `Insert`, have no use for their bodies, which they discard. Each component declares in its own specification (the `allow-body` attribute of the `<component-specification>`) whether is allows or discards its body.

In the previous example, the `Insert` component had a body, the text "Joe User". This supports WYSI-

2 More correct would be to say "its container's template" as a component may be contained within another component. For simplicities sake, we'll describe this as if it was always a simple two-level heirarchy even though practical Tapestry applications can be many levels deep.

WYG preview; the text will be displayed when previewing. Since the `Insert` component discards its body, this text will not be used at runtime, instead the OGNL expression `user.name` will be evaluated and the result inserted into the response.

No components in discarded blocks

If you put a component inside the body of an `Insert` (or any other component that discards its body), then Tapestry will throw an exception. You aren't allowed to create a component simply to discard it.

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 component would be "insert".

Providing explicit ids for your components is rarely required, but often beneficial. It is especially useful for form control components,

Each component may only appear *once* in the template. You simply can't use the same component repatedly ... 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

Component parameters may always be specified in the page or component specification, using the `<binding>`, `<static-binding>` and `<message-binding>` elements. Prior to Tapestry 3.0, that was the only way ... but with 3.0, it is possible to specify parameters directly within the HTML template.

Using either style of component (declared or implicit), parameters of the component may be *bound* by adding attributes to the tag. Most attributes bind parameters to a static (unchanging) value, equivalent to using the `<static-binding>` element in the specification. Static bindings are just the literal text, the attribute value from the HTML template.

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. Every page, and every component, is allowed to have its own set of messages (stored in a set of `.properties` files), and the `message:` prefix allows access to the localized messages stored in the files.

Seperation of Concerns

Before Tapestry 3.0, there was a more clear separation of concerns. The template could only have declared components (not implicit), and any informal attributes in the template were always static values. The type of the component and all its formal parameters were always expressed in the specification. The template was very much focused on presentation, and the specification was very much focused on business logic. There were always minor exceptions to the rules, but in general, seperation of concerns was very good.

With Tapestry 3.0, you can do more in the HTML template, and the specification file is much less important ... but the seperation of concerns is much more blurred together. It is very much acceptible to mix and match these approaches, even within a single page. In general, when learning Tapestry, or when prototyping, it is completely appopriate to do as much as possible in the HTML template. For large and complex applications, there are benefits to moving as much of the dynamic logic as possible out of the template and into the specification.

Formal and informal parameters

Components may accept two types of parameters: *formal* and *informal*. Formal parameters are those defined in the component's 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 (the list of which expands all the time).

If you are used to developing with JSPs and JSP tags, this will be quite a difference. JSP tags have the equivalent of formal parameters (they are called "tag attributes"), but nothing like informal parameters. Often a relatively simply JSP tag must be bloated with dozens of extra attributes, to support arbitrary HTML attributes.

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.

Another conflict can occur when the HTML template specified an attribute that the component needs to render itself. For example, the `DirectLink` component generates a `<a>` tag, and needs to control the `href` attribute. However, for preview purposes, it often will be written into the HTML template as:

```
<a jwcid="@DirectLink" listener=". . ." href="#"> . . . </a>
```

This creates a conflict: will the template `href` be used, or the dynamically generated value produced by

the `DirectLink` component, or both? The answer is: the component wins. The `href` attribute in the template is ignored.

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. Case is ignored when comparing attribute names to reserved names.

Template directives

For the most part, a Tapestry page or component template consists of just static HTML intermixed with tags representing components (containing the `jwcid` attribute). The overarching goal is to make the Tapestry extensions completely invisible.

Tapestry supports a limited number of additional directives that are not about component placement, but instead address other concerns about integrating the efforts of HTML developers with the Java developers responsible for the running application.

Localization

Tapestry includes a number of localization features. An important part of which is to allow each page or component to have its own catalog of localized messages (modeled after the Java `ResourceBundle` class).

The page (or component) message catalog is a collection of `.properties` files that are stored with the page or component specification. They follow the same naming conventions as for `ResourceBundle`s, so component `MyComponent` (whose specification file is `MyComponent.jwc`) might have a default message file of `MyComponent.properties`, and a French translation as `MyComponent_fr.properties`.

No global message catalog

On oft-requested feature for Tapestry is to have a global message catalog, and a way to access that catalog from the individual pages and components. This would allow common messages to be written (and translated) just once. This is a feature that may be added to Tapestry 3.1.

As we've seen, it is possible to access the messages for a page or component using the `message:` prefix on a component parameter (or use the `<message-binding>` element in a page or component specification).

What about the static text in the template itself? How does that get translated? One possibility would be to make use of the Insert component for each piece of text to be displayed, for example:

```
<span jwcid="@Insert" value="message:hello">Hello</span>
```

This snippet will get the `hello` message from the page's message catalog and insert it into the response. The text inside the `<span>` tag is useful for WYSIWYG preview, but will be discarded at runtime in favor of a message string from the catalog, such as "Hello", "Hola" or "Bonjour" (depending on the selected locale).

Because, in an internationalized application, this scenario will occur with great frequency, Tapestry includes a special directive to perform the equivalent function:

```
<span key="hello">Hello</span>
```

This is not an `Insert` component, but behaves in a similar way. The tag used must be `<span>`. You do not use the `message:` prefix on the message key (`hello`). You can't use OGNL expressions.

Normally, the `<span>` does not render, just the message. However, if you specify any additional attributes in the `<span>` tag (such as, commonly, `id` or `class` to specify a CSS style), then the `<span>` will render around the message> For example, the template:

```
<span class="error" key="invalid-access">Invalid Access</span>
```

might render as:

```
<span class="error">You do not have the necessary access.</span>
```

In this example, the placeholder text "Invalid Access" was replaced with a much longer message acquired from the message catalog.

In rare cases, your message may have pre-formatted HTML inside it. Normally, output is filtered, so that any reserved HTML characters in a message string are expanded to HTML entities. For example, a < will be expanded to `&lt;`. If this is not desired, add `raw="yes"` to the `<span>`. This defeats the filtering, and text in the message is passed through as-is.

$remove$ jwcid

HTML templates in Tapestry serve two purposes. On the one hand, they are used to dynamically render pages that end up in client web browsers. On the other hand, they allow HTML developers to use WYSIWYG editors to modify the pages without running the full application.

We've already seen two ways in which Tapestry accomidates WYSIWYG preview. Informal component parameters may be quietly dropped if they conflict with reserved names defined by the component. Components that discard their body may enclose static text used for WYSIWYG prefix.

In some cases, we need even more direct control over the content of the template. Consider, for example, the following HTML template:

Example 2.2. HTML template with repetative blocks (partial)

```
<table>
  <tr>
    <th>First Name</th>
    <th>Last Name</th>
  </tr>
  <tr jwcid="loop">
    <td><span jwcid="insertFirstName">John</span></td>
    <td><span jwcid="insertLastName">Doe</span></td>
  </tr>
  <tr>
    <td>Frank</td>
    <td>Smith</td>
  </tr>
  <tr>
    <td>Jane</td>
    <td>Jones</td>
```

```
  </tr>
</table>
```

This is part of the HTML template that writes out the names of a list of people, perhaps from some kind of database. When this page renders, the `loop` component (presumably a `Foreach`, such details being in the page's specification) will render its body zero or more times. So we might see rows for "Frank Miller", "Alan Moore" and so forth (depending on the content of the database). However, every listing will also include "Frank Smith" and "Jane Jones" ... because the HTML developer left those two rows in, to ensure that the layout of the table was correct with more than one row.

Tapestry allows a special `jwcid`, `$remove$`, for this case. A tag so marked is not a component, but is instead eliminated from the template. It is used, as in this case, to mark sections of the template that are just there for WYSIWYG preview.

> Normally, `$remove$` would not be a valid component id, because it contains a dollar sign.

With this in mind, the template can be rewritten:

Example 2.3. Updated HTML template (partial)

```
<table>
  <tr>
    <th>First Name</th>
    <th>Last Name</th>
  </tr>
  <tr jwcid="loop">
    <td><span jwcid="insertFirstName">John</span></td>
    <td><span jwcid="insertLastName">Doe</span></td>
  </tr>
  <tr jwcid="$remove$">
    <td>Frank</td>
    <td>Smith</td>
  </tr>
  <tr jwcid="$remove$">
    <td>Jane</td>
    <td>Jones</td>
  </tr>
</table>
```

With the `$remove$` blocks in place, the output is as expected, one row for each row read from the database, and "Frank Smith" and "Jane Jones" nowhere to be seen.

No components in removed blocks

It's not allowed to put components inside a removed block. This is effectively the same rule that prevents components from being put inside discarded component bodies. Tapestry will throw an exception if a template violates this rule.

`$content$ jwcid`

In Tapestry, components can have their own templates. Because of how components integrate their own

templates with their bodies (the portion from their container's template), you can do a lot ofn iteresting things. It is very common for a Tapestry application to have a Border component: a component that produces the `<html>`, `<head>`, and `<body>` tags (along with additional tags to reference stylesheets), plus some form of navigational control (typically, a nested table and a collection of links and images).

Once again, maintaining the ability to use WYSIWYG preview is a problem. Consider the following:

```
<html>
  <head>
    <title>Home page</title>
    <link rel="stylesheet" href="style.css" type="text/css">
   </head>
 <body>

   <span jwcid="border">

     <!-- Page specific content: -->

     <form jwcid=". . .">
       . . .
     </form>

   </span>
</body>
```

It is quite common for Tapestry applications to have a *Border* component, a component that is used by pages to provide the `<html>`, `<head>`, and `<body>` tags, plus common navigational features (menus, copyrights, and so forth). In this example, it is presumed that the `border` component is a reference to just such as component.

When this page renders, the page template will provide the `<html>`, `<head>` and `<body>` tags. Then when the `border` component renders, it will *again* render those tags (possibly with different attributes, and mixed in to much other stuff).

If we put a `$remove$` on the `<html>` tag in the page template, the entire page will be removed, causing runtime exceptions. Instead, we want to identify that the portion of the template *inside* the `<body>` tag (on the page template) is the only part that counts). The `$content$` component id is used for this purpose:

```
<html>
  <head>
    <title>Home page</title>
    <link rel="stylesheet" href="style.css" type="text/css">
   </head>
 <body jwcid="$content$">

   <span jwcid="border">

     <!-- Page specific content: -->

     <form jwcid=". . .">
       . . .
     </form>

   </span>
</body>
```

The `<body>` tag, the text preceding its open tag, the `</body>` tag, and the text following it are all re-

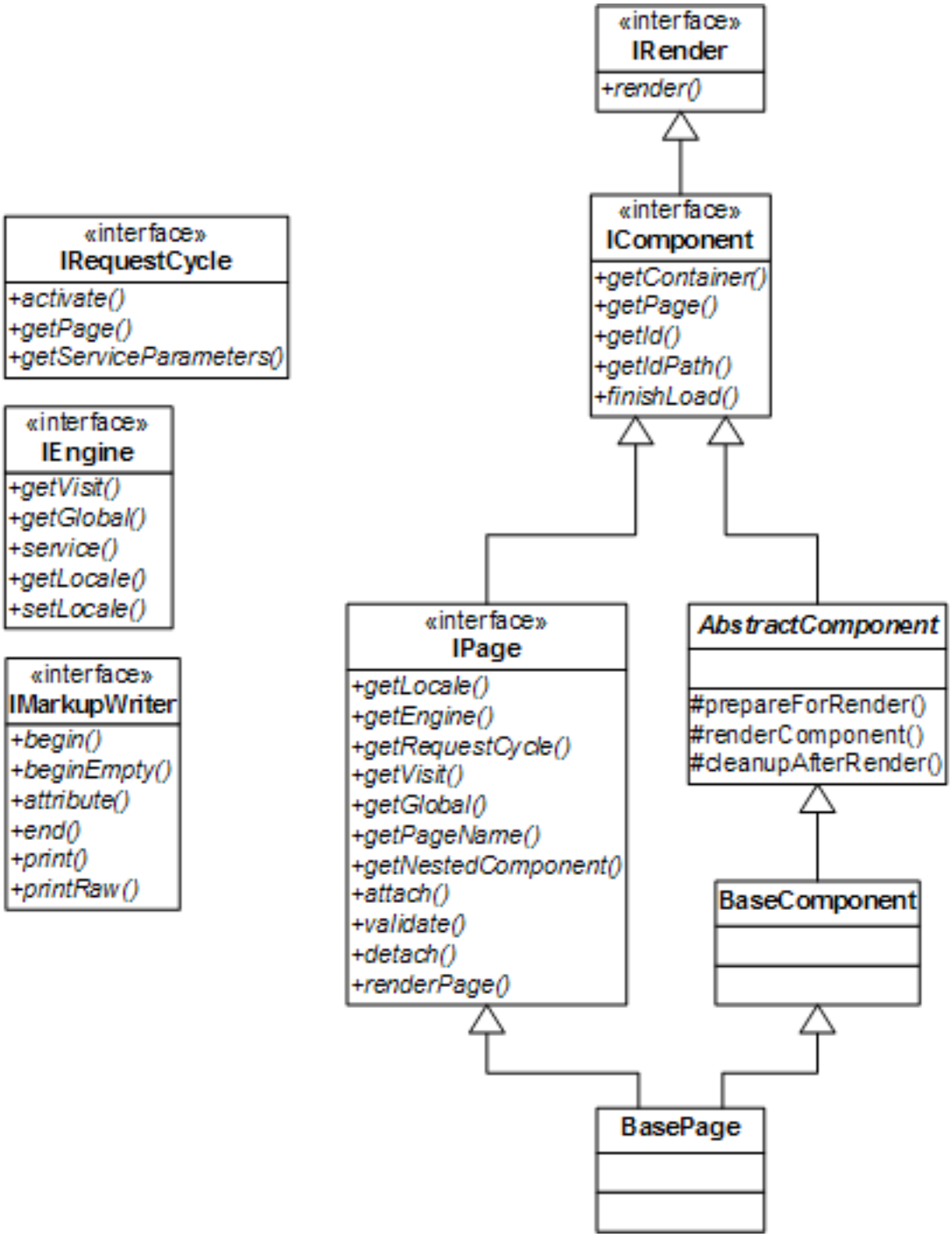
moved. It's as if the template consisted only of the `<span>` tag for the `border` component.

Chapter 3. Creating Tapestry components

Introduction

Tapestry is a component based web application framework; components, objects which implement the `IComponent` interface, are the fundamental building blocks of Tapestry. Additional objects, such as the the engine, `IMarkupWriter` and the request cycle are infrastructure. The following figure identifies the core Tapestry classes and interfaces.

Figure 3.1. Core Tapestry Classes and Interfaces



Tapestry components can be simple or complex. They can be specific to a single application or completely generic. They can be part of an application, or they can be packaged into a component library.

All the techniques used with pages work with components as well ... pages are a specialized kind of Tapestry component. This includes specified properties (including persistent properties) and listener methods.

Components fit into the overall page rendering process because they implement the `IRender` interface. Components that inherit from `BaseComponent` will use an HTML template. Components that inherit

from `AbstractComponent` will render output in Java code, by implementing method `renderComponent()`.

The components provided with the framework are not special in any way: they don't have access to any special APIs or perform any special down-casts. Anything a framework component can do, can be done by your own components.

Component Specifications

Every component has a component specification, a file ending in `.jwc` whose root element is `<component-specification>`.

Each component's specification defines the basic characteristics of the component:

- The Java class for the component (which defaults to `BaseComponent`)
- Whether the component uses its body, or discards it (the `allow-body` attribute, which defaults to `yes`)
- The name, type and other information for each *formal* parameter.
- Whether the component allows informal parameters or discards them (the `allow-informal-parameters` attribute, which defaults to `yes`)
- The names of any *reserved parameters* which may *not* be used as informal parameters.

Beyond those additions, a component specification is otherwise the same as a `<page-specification>`.

When a component is referenced in an HTML template (using the *`@Type`* syntax), or in a specification (as the `type` attribute of a `<component>` element), Tapestry must locate and parse the component's specification (this is only done once, with the result cached for later).

Tapestry searches for components in the following places:

- As specified in a `<component-type>` element (with the application specification)
- In the same folder (typically, `WEB-INF`) as the application specification
- In the `WEB-INF/`*`servlet-name`* folder (*`servlet-name`* is the name of the Tapestry `ApplicationServlet` for the application) 3
- In the `WEB-INF` folder
- In the root context directory

Generally, the *correct* place is in the `WEB-INF` folder. Components packaged into libraries have a different (and simpler) search.

Coding components

When creating a new component by subclassing `AbstractComponent`, you must write the `renderComponent()` method. This method is invoked when the components container (typically, but not

3 This is a very rare option that will only occur when a single WAR file contains multiple Tapestry applications.

always, a page) invokes its own `renderBody()` method.

```
protected void renderComponent(IMarkupWriter writer, IRequestCycle cycle)
{
  . . .
}
```

The `IMarkupWriter` object is used to produce output. It contains a number of `print()` methods that output text (the method is overloaded for different types). It also contains `printRaw()` methods -- the difference being that `print()` uses a filter to convert certain characters into HTML entities.

`IMarkupWriter` also includes methods to simplify creating markup style output: that is, elements with attributes.

For example, to create a `<a>` link:

```
writer.begin("a");
writer.attribute("url", url);
writer.attribute("class", styleClass);

renderBody(writer, cycle);

writer.end(); // close the <a>
```

The `begin()` method renders an open tag (the `<a>`, in this case). The `end()` method renders the corresponding a. As you can see, writing attributes into the tag is very simple.

The call to `renderBody()` is used to render *this* component's body. A component doesn't have to render its body; the standard `Image` component doesn't render its body (and its component specification indicates that it discards its body). The `Conditional` component decides whether or not to render its body, and the `Foreach` component may render its body multiple times.

A component that allows informal parameters can render those as well:

```
writer.beginEmpty("img");
writer.attribute("src", imageURL);
renderInformalParameters(writer, cycle);
```

This example will add any informal parameters for the component as additional attributes within the `<img>` element. These informal parameters can be specified in the page's HTML template, or within the `<component>` tag of the page's specification. Note the use of the `beginEmpty()` method, for creating a start tag that is not balanced with an end tag (or a call to the `end()` method).

Component Parameters

A Tapestry page consists of a number of components. These components communicate with, and coordinate with, the page (and each other) via *parameters*.

A component parameter has a unique name and a type (a Java class, interface, or primitive type name). The `<parameter>` component specification element is used to define formal component parameters.

In a traditional desktop application, components have *properties*. A controller may set the properties of a component, but that's it: properties are write-and-forget.

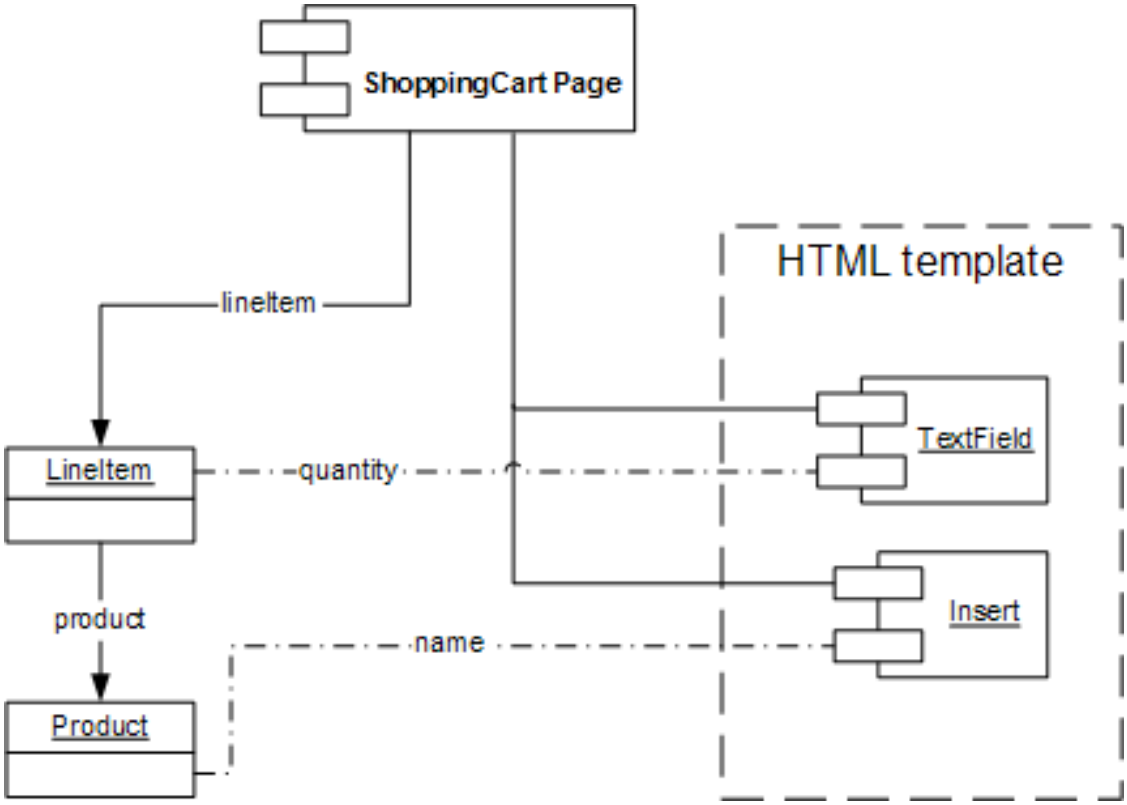
The Tapestry model is a little more complex. A component's parameters are *bound* to properties of the

enclosing page. The component is allowed to read its parameter, to access the page property the parameter is bound to. A component may also *update* its parameter, to force a change to the bound page property.

The vast majority of components simply read their parameters. Updating parameters is more rare; the most common components that update their parameters are form control components such as `TextField` or `Checkbox`.

Because bindings are in the form of OGNL expressions, the property bound to a component parameter may not directly be a property of the page ... using a property sequence allows great flexibility.

Figure 3.2. Parameter Bindings



Using OGNL, the `TextField` component's `value` parameter is bound to the `LineItem`'s `quantity` property, using the OGNL expression `lineItem.quantity`, and the `Insert` component's `value` parameter is bound to the `Product`'s `name` property using the OGNL expression `lineItem.product.name`.

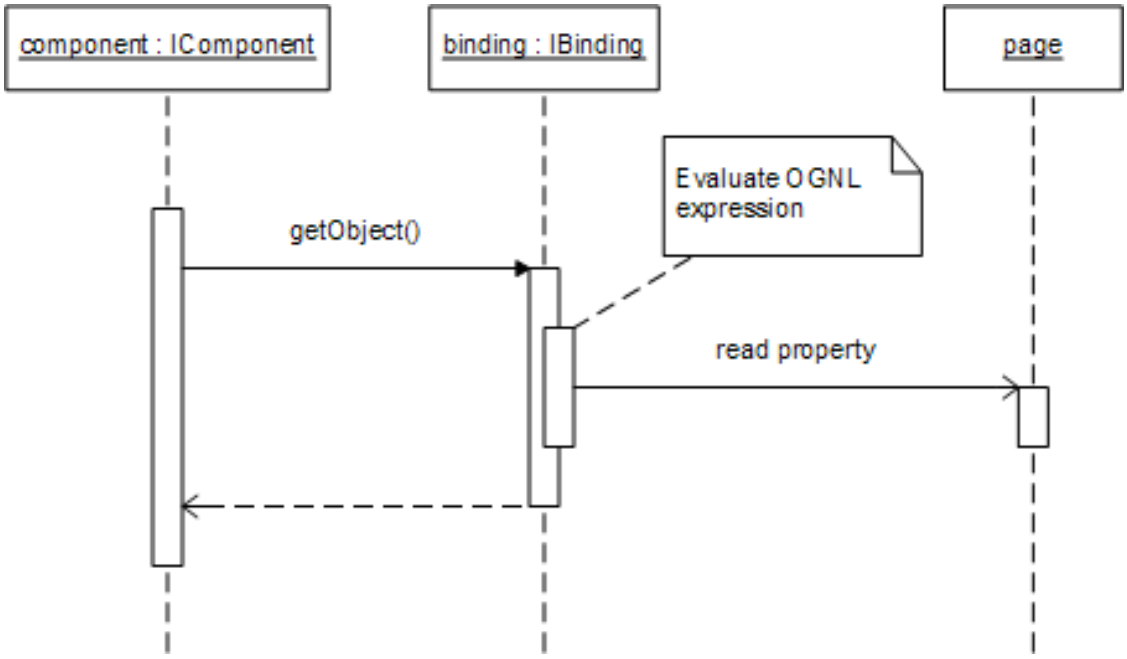
Not all parameter bindings are writable. So far, the examples have been for parameters bound using the `<binding>` specification element (or the equivalent use of the `ognl:` prefix in an HTML template). *Invariant bindings* are also possible--these are bindings directly to fixed values that never change and can't be updated. The `<static-binding>` element is invariant; it's HTML template equivalent is a attribute with no prefix. Likewise, the `<message-binding>` element, and the `message:` prefix on an attribute, are invariant.

Using Bindings

To understand how Tapestry parameters work, you must understand how the binding objects work (even

though, as we'll see, the binding objects are typically hidden). When a component needs access to a bound parameter value, it will invoke the method `getObject()` on `IBinding`

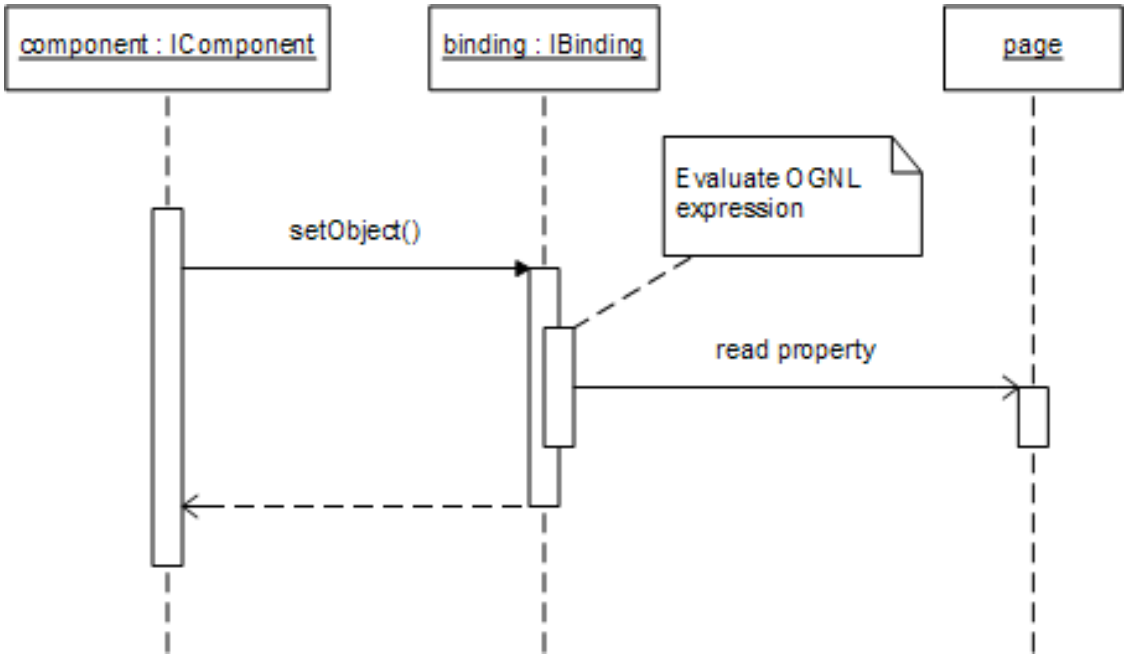
Figure 3.3. Reading a Parameter



The `getObject()` method on `IBinding` will (if the binding is dynamic) evaluate the OGNL expression (provided in the `<binding>` specification element) to access a property of the page. The result is that cast or otherwise coerced to a type useful to the component.

Updating a parameter is the same way, except that the method is `setObject()`. Most of the implementations of `IBinding` (those for literal strings and localize messages), will throw an exception immediately, since they are invariant.

Figure 3.4. Writing a Parameter



The `setObject()` method will use OGNL to update a page property.

These flows are complicated by the fact that parameters may be optional; so not only do you need to acquire the correct binding object (method `getBinding()` defined in `IComponent`), but your code must be prepared for that object to be null (if the parameter is optional).

Connected Parameter Properties

Accessing and manipulating the `IBinding` objects is tedious, so Tapestry has an alternate approach. Parameters may be represented as *connected parameter properties* that hide the existence of the binding objects entirely. If you component needs to know the value bound to a parameter, it can read the connected parameter property. If it wants to update the property bound to the parameter, the component will update the connected parameter. This is a much more natural approach, but requires a little bit of setup.

As with specified properties, Tapestry will fabricate an enhanced subclass with the necessary instance variables, accessor methods, and cleanup code.

Connected parameters are controlled by the `direction` attribute of the `<parameter>` element. [4] There are four values: `in`, `form`, `auto` and `custom`. The default is `custom`, which *does not* create a connected parameter property at all.

Direction: in

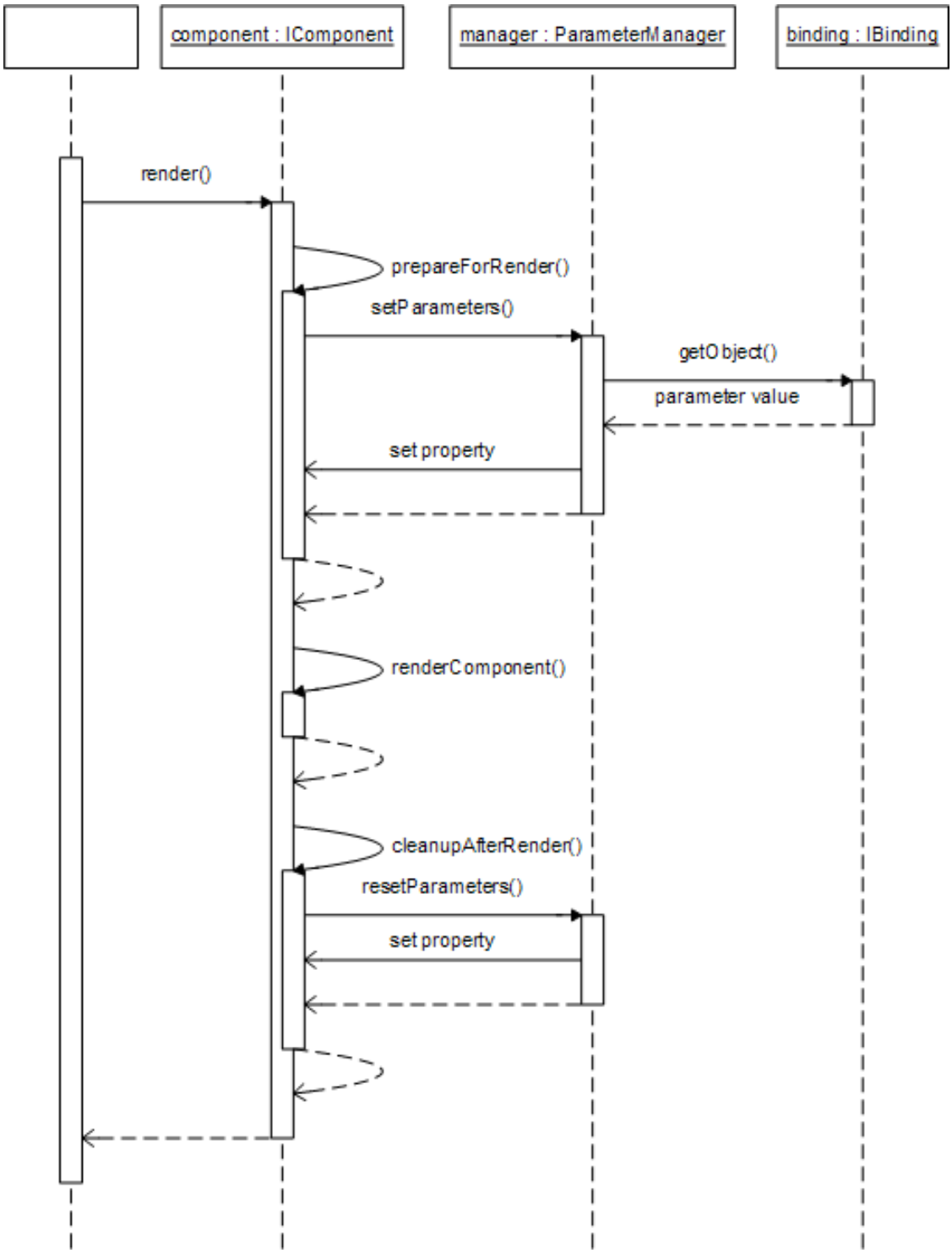
The majority of component parameters are read-only, and are only actually used within the component's `renderComponent()` method ... the method that actually produces HTML output. For such components, direction `in` is the standard, efficient choice.

The connected parameter for each component is set just before `renderComponent()` is invoked. The parameter is reset to its initial value just after `renderComponent()` is invoked.

Each component has a `ParameterManager`, whose responsibility is to set and reset connected parameter properties.

4 The name, "direction", made sense initially, but is now a bit confusing. It probably should have been called "processing" or "connection-type".

Figure 3.5. ParameterManager and `renderComponent()`



The `ParameterManager` will read the values bound to each parameter, and update the connected parameter property before the component's `renderComponent()` method is invoked. The `ParameterManager` cleans up after `renderComponent()` is invoked.

For invariant bindings (literal strings and such), the ParameterManager will only set the connected parameter property once, and does not reset the property after `renderComponent()`.

Warning

If your component has any listener methods that need to access a parameter value, then you can't use direction `in` (or direction `form`). Listener methods are invoked outside of the page rendering process, when value stored in the connected parameter property is not set. You must use direction `auto` or `custom` in such cases.

Direction: form

Components, such as `TextField` or `Checkbox`, that produce form control elements are the most likely candidates for updating their parameters. The read a parameter (usually named `value`) when they render. When the form is submitted, the same components read a query parameter and update their `value` parameter.

The `form` direction simplifies this. For the most part, `form` is the same as `in`. The diffference is, when the form is submitted, after the component's `renderComponent()` method has been invoked, the connected parameter property is read and used to update the binding (that is, invoke the binding object's `setObject()` method).

Direction: auto

The previous direction values, `in` and `form`, have limitations. The value may only be accessed from within the component's `renderComponent()` method. That's often insufficient, especially when the component has a listener method that needs access to a parameter.

Direction `auto` doesn't use the `ParameterManager`. Instead, the connected parameter property is *synthetic*. Reading the property immediately turns around and invokes `IBinding`'s `getObject()` method. Updating the property invokes the `IBinding`'s `setObject()` function.

This can be a bit less efficient than direction `in`, as the OGNL expression may be evaluated multiple times. In Tapestry 3.0, there are other limitations: the parameter must either be an object type, or one of a limited number of primitive Java types: boolean, int or double. The parameter must also be required. Future releases of Tapestry will relax these limitations.

Removing parameter directions

Parameter directions are a bit of a sore spot: you must make too many decisions about how to use them, especially in terms of render-time-only vs. listener method. Direction `auto` is too limited and possibly too inefficient. Tapestry 3.1 should address these limitations by improving direction `auto`. Instead of specifying a direction, you'll specify how long the component can cache the value obtained from the binding object (no caching, or only while the component is rendering, or until the page finishes rendering).

Direction: custom

The `custom` direction, which is the default, *does not* create a connected parameter property. Your code is still responsible for accessing the `IBinding` object (via the `getBinding()` method of `IComponent`) and for invoking methods on the binding object.

Component Libraries

Tapestry has a very advanced concept of a *component library*. A component library contains both Tapestry components and Tapestry pages (not to mention engine services).

Referencing Library Components

Before a component library may be used, it must be listed in the application specification. Often, an application specification is *only* needed so that it may list the libraries used by the application. Libraries are identified using the `<library>` element.

The `<library>` element provides an *id* for the library, which is used to reference components (and pages) within the library. It also provides a path to the library's specification. This is a complete path for a `.library` file on the classpath. For example:

Example 3.1. Referencing a Component Library

```
<application name="Example Application">

  <library id="contrib" specification-path="/org/apache/tapestry/contrib/Contrib.l

</application>
```

In this example, `Contrib.library` defines a set of components, and those component can be accessed using `contrib:` as a prefix. In an HTML template, this might appear as:

```
<span jwcid="palette@contrib:Palette" . . . />
```

This example defines a component with id `palette`. The component will be an instance of the Palette component, supplied within the `contrib` component library. If an application uses multiple libraries, they will each have their own prefix. Unlike JSPs and JSP tag libraries, the prefix is set once, in the application specification, and is used consistently in all HTML templates and specifications within the application.

The same syntax may be used in page and component specifications:

```
<component id="palette" type="contrib:Palette">
  . . .
</component>
```

Library component search path

Previously, we described the search path for components and pages within the application. The rules are somewhat different for components and pages within a library.

Tapestry searches for library component specifications in the following places:

- As specified in a `<component-type>` element (with the library specification)
- In the same package folder as the library specification

The search for page specifications is identical: as defined in the library specification, or in the same package folder.

Using Private Assets

Often, a component must be packaged up with images, stylesheets or other resources (collectively termed "assets") that are needed at runtime. A reference to such an asset can be created using the `<private-asset>` element of the page or component specification. For example:

```
<private-asset name="logo" resource-path="images/logo_200.png"/>

<component id="image" type="Image">
  <binding name="image" expression="assets.logo"/>
</component>
```

All assets (private, context or external) are converted into instances of `IAsset` and treated identically by components (such as `Image`). As in this example, relative paths are allowed: they are interpreted relative to the specification (page or component) they appear in.

The Tapestry framework will ensure that an asset will be converted to a valid URL that may be referenced from a client web browser ... even though the actual service is inside a JAR or otherwise on the classpath, not normally referenceable from the client browser.

The *default* behavior is to serve up the *localized* resource using the asset service. In effect, the framework will read the contents of the asset and pipe that binary content down to the client web browser.

An alternate behavior is to have the framework copy the asset to a fixed directory. This directory should be mapped to a know web folder; that is, have a URL that can be referenced from a client web browser. In this way, the web server can more efficiently serve up the asset, as a static resource (that just happens to be copied into place in a just-in-time manner).

This behavior is controlled by a pair of configuration properties: `org.apache.tapestry.asset.dir` and `org.apache.tapestry.asset.URL`.

Library Specifications

A library specification is a file with a `.library` extension. Library specifications use a root element of `<library-specification>`, which supports a subset of the attributes allowed within an `<application>` element (but allowing the *same* nested elements). Often, the library specification is an empty placeholder, used to an establish a search location for page and component specifications:

```
<!DOCTYPE library-specification PUBLIC
  "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
  "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<library-specification/>
```

It is allowed that components in one library be constructed using components provided by another library. The referencing library's specification may contain `<library>` elements that identify some other library.

Libraries and Namespaces

Tapestry organizes components and pages (but *not* engine services) into *namespaces*. Namespaces are closely related to, but not exactly the same as, the library prefix established using the `<library>` element in an application or library specification.

Every Tapestry application consists of a default namespace, the application namespace. This is the namespace used when referencing a page or component without a prefix. When a page or component can't be resolved within the application namespace, the framework namespaceis searched. Only if the component (or page) is not part of the framework namespace does an error result.

In fact, it is possible to override both pages and components provided by the framework. This is frequently used to change the look and feel of the default StateSession or Exception page. In theory, it is even possible to override fundamental components such as `Insert` or `Foreach`!

Every component provides a `namespace` property that defines the namespace (an instance of `INamespace`) that the component belongs to.

You rarely need to be concerned with namespaces, however. The rare exception is when a page from a library wishes to make use of the `PageLink` or `ExternalLink` components to create a link to *another page* within the same namespace. This is accomplished (in the source page's HTML template) as:

```
<a href="#" jwcid="@PageLink" page="OtherPage" namespace="ognl:namespace"> ... <
```

Chapter 4. Managing server-side state

Server-side state is any information that exists on the server, and persists between requests. 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 an ordinary 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 object, and persistent page properties.

Understanding servlet state

Managing server-side state is one of 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 non-specific 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 [5].

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

[5] This is the replication strategy employed by BEA's WebLogic server.

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 issues.

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. 6

Tapestry doesn't mandate anything about the Visit object's class. The type of the `visit` property is `Object`. In Java code, accessing the Visit object involves a cast from `Object` to an application-specific class. The following example demonstrates how a listener method may access the visit object.

Example 4.1. Accessing the Visit object

```
public void formSubmit(IRequestCycle cycle)
{
    MyVisit visit = (MyVisit)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:

```
    MyVisit visit = (MyVisit)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

6 Another good name would have been "session", but that name is heavily overloaded throughout Java and J2EE.

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 4.2. Defining the Visit class

```
<application name="My Application">
  <property name="org.apache.tapestry.visit-class" value="mypackage.MyVisit"/>

  ...
```

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` property of 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 Global object is very similar to the 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. The Global object is specific to an individual server within a cluster; each server will have its own instance of the Global object. In a failover, the engine will connect to a new instance of the Global object within the new server.

The Global object may be accessed 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. Tapestry uses an object pool to store constructed page instances. As a page is needed, it is removed from the page pool. If there are no available pages in the pool, a fresh page instance is constructed.

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 their initial values.

The page is then returned to the page pool, where it will wait to be used in a future request. That request may be for the same end user, or for another user entirely.

Importance of resetting properties

Imagine a page containing a form in which a user enters their address and credit card information. When the form is submitted, properties of the page will be updated with the values supplied by the user. Those values must be cleared out before the page is stored into the page pool ... if not, then the *next* user who accesses the page will see the previous user's address and credit card information as default values for the form fields!

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, even though the request may use a different page instance (from the page pool) ... or (in a clustering environment) may be handled by a completely different server.

Each persistent page property is stored individually as an `HttpSession` attribute. A call to the static method `Tapestry.fireObservedChange()` must be added to the setter method for the property (as we'll see shortly, Tapestry can write this method for you, which is the best approach). When the property is changed, its value is stored as a session attribute. Like the Servlet API, persistent properties work best with immutable objects such as `String` and Integer;. For mutable objects (including `List` and `Map`), you must be careful *not* to change the internal state of a persistent property value after invoking the setter method.

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 fabricates 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 fabricated subclass. This allows you to read and update properties inside your class, inside listener methods.

Define only what you need

You only need to define abstract accessor methods if you are going to invoke those accesor methods in your code, such as in a listener method. Tapestry will create an enhanced subclass that contains the new field, a getter method and a setter method, plus any necessary initialization methods. If you are only going to access the property using OGNL expressions, then there's no need to define either accessor method.

Transient or persistent?

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 4.3. Persistent page property: Java class

```
package mypackage;

import org.apache.tapestry.html.BasePage;

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

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

Example 4.4. 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="mypackage.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 getter and a setter.

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.

Warning

The previous paragraph may not be accurate; I believe Mindbridge may have changed this behavior recently.

This means that you may perform initialization for the property inside `finishLoad()` (instead of providing an `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.

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 4.5. Use of `initialize()` method

```
package mypackage;

import org.apache.tapestry.html.BasePage;

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 4.6. Manual persistent page property

```
package mypackage;

import org.apache.tapestry.Tapestry;
import org.apache.tapestry.html.BasePage;

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

Warning

There is very little reason to implement persistent component properties manually. Using the `<property-specification>` element is much easier.

Tapestry uses the same mechanism for persistent component properties as it does for persisting page properties (remember that pages are, in fact, specialized components). Implementing transient and persistent properties inside components involves more work than with pages as 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 interfaces, such as `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 4.7. Manual Persistent Component Properties

```
package mypackage;

import org.apache.tapestry.Tapestry;
import org.apache.tapestry.BaseComponent;
import org.apache.tapestry.event.PageDetachListener;
import org.apache.tapestry.event.PageEvent;

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 far 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 (CPU time, network bandwidth, and so forth). 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 object 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 an object 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 5. Configuring Tapestry

Requirements

Tapestry is designed to operate on a variety of different JVMs and versions of the Java Servlet API. Below you can find the list of supported and tested configurations:

Supported Java Versions

Java 1.2.2
: Operates correctly. Requires the Xerces parser to be in the classpath (usually provided by the servlet container).

Java 1.3.x
: Operates correctly. Requires the Xerces parser to be in the classpath (usually provided by the servlet container).

Java 1.4.x (recommended)
: Operates correctly.

Supported Java Servlet API Versions

Java Servlet API 2.2
: Operates correctly with minor exceptions related to character encoding of the requests due to the limitations of the Servlet API version.

Java Servlet API 2.3 (recommended)
: Operates correctly.

Web deployment descriptor

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

Example 5.1. Web Deployment Descriptor

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <distributable/> ❶
  <display-name>My Application</display-name>
  <servlet>
    <servlet-name>myapp</servlet-name> ❷
    <servlet-class>org.apache.tapestry.ApplicationServlet</servlet-class> ❸
    <load-on-startup>0</load-on-startup> ❹
  </servlet>
```

```
  <servlet-mapping>
    <servlet-name>myapp</servlet-name>
    <url-pattern>/app</url-pattern> ❺
  </servlet-mapping>

  <filter> ❻
    <filter-name>redirect</filter-name>
    <filter-class>org.apache.tapestry.RedirectFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>redirect</filter-name>
    <url-pattern>/</url-pattern>
  </filter-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 It is generally a good idea to specify `<load-on-startup>`, this causes the servlet container to instantitate and initialize the the application servlet, which in turn, reads the Tapestry application specification. Many common development errors will be spotted immediately, rather than when the first application request arrives.

❺ The servlet is mapped to `/app` within the context. The context itself has a path, determined by the application server and based on the name of the WAR file. The client web browser will see the Tapestry application as `http://`*`host/war-name`*`/app`.

Using `/app` as the URL is a common convention when creating Tapestry applications, but is not a requirement. The framework will adapt to whatever mapping you select.

❻ This filter sends a client redirect to the user when they access the web application context. The filter sends a client redirect to the user's browser, directing them to the application servlet. In this way, the "public" URL for an application can be `http://myserver/mycontext/` when, in fact, the real address is `http://myserver/mycontext/app`.

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.

The specification is normally 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 location.

If the application specification still can not be found, then an empty, "stand in" application specification is used. This is perfectly acceptible ... an application specification is typically needed only when an application makes use of component libraries, or requires some other kind of customization only possible with an application specification.

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 are 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-spec-<ification>` 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 development; 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. Unlike most other configuration values, this must be specified as a JVM system property.

`org.apache.tapestry.output-encoding`

Defines the character set used by the application to encode its HTTP responses. This is also the character set that the application assumes that the browser uses when submitting data unless it is not specified differently in the HTTP request.

The default for this configuration property is UTF-8. Normally there is no need to modify this value since UTF-8 allows almost all characters to be correctly encoded and displayed.

`org.apache.tapestry.template-encoding`
Defines the character set used by the application templates. The default value is ISO-8859-1.

Please see the Character Sets section for more information.

Application extensions

Tapestry is designed for flexibility; this extends beyond simply configuring the framework, and encompasses actually replacing or augmenting the implementation of the framework. If Tapestry doesn't do what you want it to, there are multiple paths for extending, changing and overriding its normal 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.

Your application may have its own set of extensions not related to Tapestry framework extension points. For example, you might have an application extension referenced from multiple pages to perform common operations such as JNDI lookups.

You may access application extensions via the engine's specification property. For example:

```
IEngine engine = getEngine();
IApplicationSpecification specification = engine.getSpecification();

myExtension myExtension = (MyExtension) specification.getExtension("myExtension");
```

Each application extension used with an framework extension point 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).

`org.apache.tapestry.ognl-type-converter`
: Specifies an implementation of ognl.TypeConverter to be used for expression bindings. See OGNL's Type Converter documentation for further information on implementing a custom type converter.

Character Sets

Tapestry is designed to make the web application localization easy and offers the ability to define different localized templates for the same component. For example, `Home.html` would be the default template of the Home page, however `Home_fr.html` would be used in all French locales, while `Home_zh_CN.html` would be used in China and `Home_zh_TW.html` would be used in Taiwan.

Web developers and designers in different countries tend to use different character sets for the templates they produce. English, German, French templates are typically produced in ISO-8859-1, Russian templates often use KOI8-R, and Chinese texts are normally written in Big5. Tapestry allows the application to configure the character set used in its templates and makes it possible to use different character sets for templates associated with different components and different locales.

The character set of a template is defined using the `org.apache.tapestry.template-encoding` configuration property. The search path of this property is slightly different then the standard one and allows specific components to use other character sets:

- As a `<property>` of the `<page-specification>` or the `<component-specification>` (in the page or component specification).

  This configuration will apply only to the page or component where it is defined.

- As a `<property>` of the `<library-specification>` (in the library specification, if the

components are included in a library).

This configuration will apply to all pages and components in the library.

- 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.
- The hard-coded default "ISO-8859-1".

Tapestry also makes it possible to define the character set used by the templates specific to a particular locale by appending the locale to the property name above. As an example, the `org.apache.tapestry.template-encoding_ru` configuration property would define the character set used by the Russian templates, such as `Home_ru.html`. This allows templates for different locales to use different character sets, even though they are in the same application. For example, it is possible for all Russian templates in the application to use the KOI8-R character set and all Chinese templates to use Big5 at the same time.

The character sets used by the templates do not reflect in any way on the character set Tapestry uses to encode its response to the browser. The character sets are used when reading the template to translate it appropriately into Unicode. The output character set is defined by the `org.apache.tapestry.output-encoding` configuration property.

Appendix A. Tapestry Object Properties

When using Tapestry, an important aspect of your work is to leverage the properties exposed by the various objects within Tapestry. A page has properties (inherited from base classes such as `AbstractComponent` and `BasePage`) and contains components and other objects with more properties. Pages are connected to an engine, which exports its own set of properties. This appendix is a quick guide to the most common objects and their properties.

Table A.1. Tapestry Object Properties

Property name	Defining class	Property type	Description
activePageNames	`BaseEngine`	`Collection` of `String`	Names of all pages for which a page recorder has been created.
assets	`IComponent`	`Map` of `IAsset`	Localized assets as defined in the component's specification.
beans	`IComponent`	`IBeanProvider`	Used to access beans defined using the `<bean>` specification element.
bindingNames	`IComponent`	`Collection` of `String`	The names of all formal and informal parameter bindings for the component.
bindings	`IComponent`	`Map` of `IBinding`	All bindings (for both formal and informal parameters) for this component, keyed on the parameter name.
body	`AbstractComponent`	`IRender`[]	The body of the component: the text (as `IRender`) and components (which inherit from `IRender`) that the component directly encloses within its container's template.
bodyCount	`AbstractComponent`	int	The active number of elements in the `body` property array.
componentClassEnhancer	`IEngine`	`IComponentClassEnhancer`	Object responsible for dynamic creation of enhanced subclasses of Tapestry pages and components.
components	`IComponent`	`Map` of `IComponent`	All components contained by this component, keyed on the component id.

Property name	Defining class	Property type	Description
contextPath	`IEngine`	`String`	The path, if any, for the web application context.
changeObserver	`IPage`	`ChangeObserver`	An object that recieves notifications about changes to persistent page properties.
componentMessagesSource	`IEngine`	`IComponentMessagesSource`	An object that allows components to find their set of localized messages.
container	`IComponent`	`IComponent`	The page or component which contains this component. Pages will return null.
dataSqueezer	`IEngine`	`DataSqueezer`	Object used to encode and decode arbitrary values into a URL while maintaining their type.
dirty	`AbstractEngine`	boolean	True if the engine has been (potentially) modified, and should be stored into the `HttpSession`.
disabled	`IFormComponent`	boolean	If true, the component should be disabled (and not respond to query parameters passed up in the request).
displayName	`IFormComponent`	`String`	Localized string to be displayed as a label for the form control. Most implementations leave this undefined (as null).
engine	`IPage`	`IEngine`	The engine to which the page is currently attached.
extendedId	`IComponent`	`String`	An "extended" version of the `idPath` property that includes the name of the page containing the component as well.
form	`IFormComponent`	`IForm`	The form which encloses the form control component.
global	`IEngine`, `IPage`	`Object`	The Global object for the application.
hasVisit	`AbstractEngine`	boolean	Returns true if the Visit object has been created, false initially.
id	`IComponent`	`String`	The id of the component, which is unique within its container. In many

Property name	Defining class	Property type	Description
			cases, the framework may have assigned an automatically generated id. Pages do not have an id and return null.
idPath	`IComponent`	`String`	A sequence of id's used to locate a component within a page. A component `bar` within a component `foo` within a page will have an `idPath` of `foo.bar`. Pages return null.
listeners	`AbstractComponent`, `AbstractEngine`	`ListenerMap`	Used to map listener methods as objects that implement the `IActionListener` interface.
locale	`IEngine`	`Locale`	The locale for the current client; this is used when loading pages from the page pool, or when instantiating new page instances.
locale	`IPage`	`Locale`	The locale to which the page and all components within the page is localized.
location	*many*	`ILocation`	The location that should be used with any error messages generated about the object. This is ultimately the file, line (and even column) of the template or specification file responsible for defining the object (be it a component, a page, or some other kind of object).
messages	`IComponent`	`IMessages`	Localized messages for the component.
name	`IFormComponent`	`String`	The name, or element id, assigned to the form control by the `IForm`. This is set as the component renders (but the property can then be read after the component renders).
namespace	`IComponent`	`INamespace`	The namespace containing the component. Components are always within *some* namespace,

Property name	Defining class	Property type	Description
			whether it is the default (application) namespace, the framework namespace, or a namespace for a component library.
outputEncoding	`AbstractPage`	`String`	Output encoding for the page.
page	`IComponent`	`IPage`	The page which ultimately contains the component.
propertySource	`IEngine`	`IPropertySource`	Source for configuration properties.
pageName	`IPage`	`String`	The fully qualified page name (possibly including a namespace prefix).
pageSource	`IEngine`	`IPageSource`	The object used to obtain page instances.
pool	`IEngine`	`Pool`	Stores objects that are expensive to create.
requestCycle	`IPage`	`IRequestCycle`	The request cycle to which the page is currently attached.
resetServiceEnabled	`IEngine`	boolean	If true, the reset service is enabled. The reset service is disabled by default.
resourceResolver	`IEngine`	`IResourceResolver`	Object responsible for locating classes and classpath resources.
scriptSource	`IEngine`	`IScriptSource`	Object that parses and caches script specifications.
servletPath	`IEngine`	`String`	The URL path used to reference the application servlet (including the context path, if any).
specification	`IComponent`	`IComponentSpecification`	The specification which defines this component. Often used to access meta data defined in the component's specification using the `prop-<erty>` element.
specification	`IEngine`	`IApplicationSpecification`	The specification for the application.
specificationSource	`IEngine`	`ISpecificationSource`	Object responsible for reading and caching page and component specifications.
stateful	`IEngine`	boolean	If true, then an `HttpSession` has

Property name	Defining class	Property type	Description
			been created for the client to store server-side state. Initially false.
templateSource	`IEngine`	`ITemplateSource`	Object responsible for reading and caching page and component templates.
visit	`IEngine`	`Object`	Returns the Visit object for the current client, or null if the Visit object has not yet been created.
visit	`IPage`	`Object`	Returns the Visit object for the current client, creating it if necessary.

Appendix B. Tapestry JAR files

The Tapestry distribution includes the Tapestry JARs, plus all the dependencies (other libraries that Tapestry makes use of). The JAR files are in the `lib` folder (or in folders beneath it).

`tapestry-3.0.jar`
: The main Tapestry framework. This is needed at compile time and runtime. 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.

`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.

`ext/*.jar`
: Frameworks needed when compiling the framework and at runtime. This is several other Jakarta frameworks (including BSF and BCEL), plus the OGNL and Javassist frameworks.

`j2ee/*.jar`
: Contains the J2EE and Servlet APIs. These are needed when building the framework, but are typically provided at runtime by the servlet container or application server.

Appendix C. Tapestry Specification DTDs

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

Table C.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 C.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 `IEngine` to instantiate. Defaults to `BaseEngine` if not specified.

Figure C.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 C.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 C.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 C.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 C.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 C.7. <component> Elements

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

<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 C.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 C.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.

Name	Type	Required ?	Default Value	Description
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 parameter) 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 C.10. `<component-specification>` Elements

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

`<configure>` element

Appears in: `<extension>`

Allows a JavaBeans property of the extension to be set from a statically defined value. The `config-<ure>` 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 C.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 C.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 C.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 constructor.
immediate	`yes|no`	no	`no`	If `yes`, the extension is instantiated when the icaspeciftion is read. If `no`, then the extension is not created until first needed.

Figure C.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 C.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 C.16. `<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 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 C.17. `<library>` Attributes

Name	Type	Required ?	Default Value	Description
id	string	yes		The id associated with the library.

Name	Type	Required ?	Default Value	Description
				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 C.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 C.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 BSF-supported 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 C.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 C.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

Name	Type	Required ?	Default Value	Description
				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 C.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 C.23. <page-specification> Elements

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

<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 C.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 through the binding. `in` means the prop-

Name	Type	Required ?	Default Value	Description
				erty 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 C.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 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 C.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 C.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 (int, boolean, etc.). It may be suffixed with [] to indicate an array of the indicated type.
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 C.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 C.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 C.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 contain-

Name	Type	Required ?	Default Value	Description
				ing 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 C.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 C.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 prob-

Name	Type	Required ?	Default Value	Description
				lematic punctuation).

Appendix D. 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 D.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 D.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

Name	Type	Required ?	Default Value	Description
				each successive index.

Figure D.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 D.4. `<if>` Attributes

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

Figure D.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 D.6. `<if-not>` Attributes

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

Figure D.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 D.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 D.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 D.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 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 D.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 D.12. `<let>` Elements

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

<script> element

Root element

The root element of a Tapestry script specification.

Figure D.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 D.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 D.15. <unique> Elements

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