Component Templates

Under Tapestry, a **component template** is a file that contains the markup for a component.

Component templates are *well formed XML documents*. That means that every open tag must have a matching close tag, every attribute must be quoted, and so forth.

Note: At runtime, Tapestry parses the documents and only checks for wellformedness. Even when the document has a DTD or schema, there are no validity checks.

For the most part, these templates are standard HTML/XHTML; Tapestry extensions to ordinary markup are provided in the form of a Tapestry XML namespace (xmlns):

A template for a page

```
<html t:type="layout" xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
    <h1>Bonjour from HelloWorld component.</h1>
</html>
```

We'll cover the specific content of templates shortly, but first a few details about connecting a component to its template.

Related Articles

- Content Type and Markup
- Component Reference
- Component Libraries
- Component Classes
- Templating and Markup FAQ
- Request Processing
- Configuration
- Component Parameters
- Assets

Template Location

A component template shares the same name as its corresponding class file, but with a ".tml" ending (i.e., **T**apestry **M**arkup **L**anguage), and is stored in the same package as the corresponding component class.

Under a typical Maven directory structure, the Java class and template files for a *component* might be:

Java class:	`src/main/java/org/example/myapp/components/MyComponent.java`
Template:	`src/main/resources/org/example/myapp/components/MyComponent.tml`

Likewise, the Java class and template files for a *page* might be:

Java class:	`src/main/java/org/example/myapp/pages/MyPage.java`
Template:	`src/main/resources/org/example/myapp/pages/MyPage.tml`

The template and the compiled class will be packaged together in the WEB-INF/classes folder of the application WAR.

For *pages* (but not other components), a second location will be searched: in the web application context. The location is based on the logical name of the page, in the previous example, the template would be `MyPage.tml` in the root folder of the web application.

A template on the classpath takes precedence over a file in the web application context.

Allowing pages to store their template in the web context is a feature that may go away at some point. It was included as a way for HTML designers to edit template directly and live preview the template directly, without executing the Tapestry application. This comes with a large number of limitations and leads to a false sense of security that a template that previews correctly will render properly (this is not always the case).

Template Localization

Main Article: Localization

Templates are handled in much the same way as individual files of a component's message catalog: the effective locale is inserted into the name of the file. Thus a German users will see the content generated from `MyPage_de.tml` and French users will see content generated from `MyPage_fr.tml`. When no specific localization is available, the default location (`MyPage.tml`) is used.

It is necessary to enable support for a locale before Tapestry will attempt to localize to that locale. This requires configuration in your application module (usually AppModule.java); if you are using the Tapestry Quickstart archetype, only locale "en" will be enabled by default.

Template Doctypes

As mentioned above, component templates are well-formed XML documents. This means that if you want to use any Named HTML entities (such as & < > ©), you must use an HTML or XHTML doctype in your template *(starting in 5.3, this is more-or-less automatic, see notes below)*. If you choose to use (X)HTML doctypes in your templates, they will be passed on to the client in the resultant (X)HTML. Note that if your pages are composed of multiple components, each with a template, and each template contains a doctype declaration, only the first doctype encountered by the template parser will be passed on to the client.

It should also be noted that even though **X**HTML DTDs are valid XML DTDs, HTML DTDs aren't. This means that HTML doctypes cannot be used by XML parsers. Tapestry works around this limitation internally by using XHTML DTDs to parse templates that use HTML DTDs. This internal mapping is possible because XHTML 1.0 is nothing more than "a reformulation of the three HTML 4 document types as applications of XML 1.0," as per the W3C. Don't worry though – the original HTML 4 doctype will still be emitted to the client!

The following are the most common (X)HTML doctypes:

```
<!DOCTYPE html>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
   "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
   "http://www.w3.org/TR/html4/strict.dtd">

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
   "http://www.w3.org/TR/html4/loose.dtd">
```

The first one is for HTML5 and is recommended for Tapestry 5.2.5 and later. In versions prior to Tapestry 5.2.5, Tapestry didn't support the HTML5 doctype directly (but see the comments at TAP5-1040 for a work-around).

> **Added in 5.3**
>
> Tapestry 5.3 introduced two significant improvements to template Doctypes. A template without a is parsed as if it had the HTML Doctype ({{}}). In fact, Tapestry creates an in-memory copy of the template that includes the doctype. A template with the HTML Doctype ({{}}) is parsed *as if* it had the XHTML transitional Doctype. In fact, Tapestry creates an in-memory copy of the template that replaces the line. This applies as well to a template without any Doctype, in which case the XHTML transitional Doctype is inserted at the top. In either case, this means you can use arbitrary HTML entities, such as {{©}} or {{ }} without seeing the XML parsing errors that would occur in earlier releases.

The Tapestry Namespace

Component templates should include the Tapestry namespace, defining it in the root element of the template.

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
    <head>
        <title>Hello World Page</title>
    </head>
    <body>
        <h1>Hello World</h1>
    </body>
</html>
```

This defines the namespace using the standard prefix, "t:". The examples on this page all assume the use of the standard prefix.

For backwards compatibility, you may continue to use the old namespace URIs: http://tapestry.apache.org/schema/tapestry_5_0_0.xsd or http://tapestry.apache.org/schema/tapestry_5_1_0.xsd or http://tapestry.apache.org/schema/tapestry_5_3.xsd

However, the following elements added, as part of Tapestry 5.1, will not work with the 5_0_0.xsd:

- The <t:remove> Element
- <t:content>
- <t:extension-point>
- <t:extend>
- <t:replace>

The 5_3.xsd fixes some minor bugs in the 5_1_0.xsd, but is functionally equivalent; 5_3.xsd and 5_4.xsd are identical.

Tapestry Elements

Tapestry elements are elements defined using the Tapestry namespace prefix (usually "t:").

All other elements in your templates should be in the default namespace, with no prefix (with the possible exception of any Library Namespaces (described below).

There are a certain number of Tapestry elements, listed below, that act as template directives; beyond that, any element in the Tapestry namespace will be a Tapestry component.

The <t:body> Element

In many cases, a component is designed to have its template integrate with, or "wrap around", the containing component.

The <t:body> element is used to identify where, within a component's template, its body (from the container's template) is to be rendered.

Components have control over if, and even how often, their body is rendered.

The following example is a Layout component, which adds basic HTML elements *around* the page-specific content:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_3.xsd">
    <head>
        <title>My Tapestry Application</title>
    </head>
    <body>
        <t:body/>
    </body>
</html>
```

That "<t:body/>" element marks where the containing page's content will be inserted. A page would then use this component as follow:

```
<html t:type="layout" xmlns:t="http://tapestry.apache.org/schema/tapestry_5_3.xsd"

  My Page Specific Content

</html>
```

When the page renders, the page's template and the Layout component's template are merged together:

```
<html>
  <head>
    <title>My Tapestry Application</title>
  </head>
  <body>
    My Page Specific Content
  </body>
</html>
```

Tapestry 4 users will recognize the <t:body> element as a replacement for the RenderBody component.

The <t:container> Element

A <t:container> element contains markup without being considered part of the template. This is useful for components that render several top level tags, for example, a component that renders several columns within a table row:

```
<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_3.xsd">
  <td>${label}</td>
  <td>${value}</td>
</t:container>
```

This component only makes sense when used inside a <tr> element of its container's template.

Without the <t:container> element, there would be no way to create a valid XML document as the template, because XML documents must always have a single root element.

The <t:block> Element

A <t:block> is a container of a portion of the component template. A block does not normally render; any component or contents you put inside a block will not ordinarily be rendered. However, by injecting the block you have precise control over when and if the content renders.

A block may be anonymous, or it may have an id (specified with the id attribute). Only blocks with an id may be injected into the component.

A <t:block> must be in the Tapestry namespace, but the id attribute should not be. This is different from components in the template, where the t:id attribute that defines the component id *must* be in the Tapestry namespace.

Ids must be valid Java identifiers: start with a letter, and contain only letters, numbers and underscores.

The <t:parameter> Element

This element was deprecated starting in Tapestry 5.1 and *removed* in 5.3. Use *parameter namespaces* (below) instead.

A <parameter> element is a special kind of block. It is placed inside the body of an embedded component. The block defined by the <parameter> is passed to the component. <parameter> includes a mandatory name attribute to identify which parameter of the component to bind.

The <t:content> Element

<t:content> marks a portion of the template as the actual template *content*; any markup outside the <t:content> element is ignored. This is useful for eliminating portions of the template that exist to support WYSIWYG preview of the template.

<t:content> elements may not nest.

Support for the <t:content> element was adding in Tapestry release 5.1. You must use the http://tapestry.apache.org/schema/tapestry_5_4.xsd or (or ...5_3 or ...5_1_0.xsd) namespace URI for content to be recognized (otherwise you will see an error about a missing "content" component).

<t:remove>

Marks a portion of the template for removal; it is as if the remove element and everything inside it simply was not part of the template. This is used as a kind of server-side only comment (normal HTML/XML comments are included in a page render response), or to temporarily eliminate a portion of the template. As far as Tapestry is concerned, the contents of the <remove> element do not exist (including validating consistency between components defined or injected in the Java class and the template).

Support for the <t:remove> element was added in Tapestry release 5.1. You must use the http://tapestry.apache.org/schema/tapestry_5_4.xsd or (or ...5_3 or ...5_1_0.xsd) namespace URI for remove to be recognized (otherwise you will see an error about a missing "remove" component).

Expansions

Another option when rendering output is the use of *expansions*. Expansions are special strings that may be embedded in template bodies, and borrow some syntax from the Ant build tool.

```
Welcome, ${userId}!
```

Here, `${userId}` is the expansion. In this example, the userId property of the component is extracted, converted to a string, and streamed into the output.

Expansions are allowed inside text, and inside attributes of ordinary elements, and component elements. For example:

```
<img src="${request.contextPath}/images/catalog/product_${productId}.png"/>
```

In this hypothetical example, the component class is providing a request property and a productId property, and these are being used inside the template to assemble the src attribute of the <img> element. This is component-like behavior without actual components.

Under the covers, expansions are the same as parameter bindings. The default binding prefix for expansions is "prop:" (that is, the name of a property or a property expression), but other binding prefixes are useful, especially "message:" (to access a localized message from the component's message catalog).

Do not use expansions in component parameters if the parameter's default or explicit binding prefix is "prop:" or "var:". Expansions convert the value to an immutable string, resulting in a runtime exception if the component tries to update the value. Even for read-only parameters, expansions are not as desirable, since they always convert to a string, and from there to the parameter's declared type.
Tapestry 4 users will note that expansions are a concise, easy replacement for the Insert component, and for the directive.

Note that expansions escape any HTML reserved characters. Specifically, any less-than (<), greater than (>) and ampersand (&) are replaced with <, > and & respectively. That is usually what you want. However, if your property contains HTML that you want rendered as raw markup, you can use the OutputRaw component instead, like this: `<t:OutputRaw value="someContent"/>` where `someContent` is a property containing HTML markup.

Caution: if the content comes from an untrusted source (like a public user), be sure to filter it before providing it to OutputRaw. Otherwise you've got a potential cross-site scripting vulnerability.

Embedded Components

An embedded component is identified within the template as an element in the t: namespace. Example:

```
  You have ${cartItems.size()} items in your cart.
  <t:actionlink t:id="clear">Remove All</t:actionlink>.
```

The element name, "actionlink" is used to select the type of component, ActionLink.

As elsewhere, Tapestry is insensitive to case when mapping from a component type to a component class.

Embedded components may have two Tapestry-specific parameters:

- id: A unique id for the component (within its container).
- mixins: An optional comma separated list of mixins for the the component.

These attributes are specified inside the t: namespace (i.e., `t:id="clear"`).

If the id attribute is omitted, Tapestry will assign a unique id for the element.

Non-trivial components should always be assigned a specific id, rather than letting Tapestry do it. You'll end up with shorter, more readable URLs and code that's easier to debug, because it will be more obvious how the request URL maps to your pages and components. This is even more strongly encouraged for form control components, where the component id will usually be the same as the query parameter that stores the value provided by the end user.

Ids must be valid Java identifiers: start with a letter, and contain only letters, numbers and underscores.

Any other attributes are used to bind parameters of the component. These may be formal parameters or informal parameters. Formal parameters will have a default binding prefix (usually "prop:"). Informal parameters will be assumed to be literals (i.e., the "literal:" binding prefix).

Use of the t: prefix is optional for all other attributes. Some users implement a build process where the Tapestry template files are validated ... in that case, any Tapestry-specific attributes, not defined by the underlying DTD or schema, should be in the Tapestry namespace, to avoid validation errors.

The open and close tags of a Tapestry component element define the **body** of the component. It is quite common for additional components to be **enclosed** in the body of another component:

```
<t:form>
  <t:errors/>
  <t:label for="userId"/>
  <t:textfield t:id="userId"/>
  <br/>
  <t:label for="password"/>
  <t:passwordfield t:id="password"/>
  <br/>
  <input type="submit" value="Login"/>
</t:form>
```

In this example, the <t:form> component's body contains the other components. Structurally, all of these components (the Form, Errors, Label, etc.) are peers: children of the page. They are all *embedded* within the page. The Errors, Label, TextField and so forth are *enclosed* within the Form's body ... meaning that the Form controls if, when, and under what circumstances those components will render.

In some cases, components require some kind of enclosure; for example, all of the field control components (such as TextField) will throw a runtime exception if not enclosed by a Form component.

It is possible to place Tapestry components in sub-packages. For example, your application may have a package org.example.myapp.components.ajax. Dialog. This component's normal type name is "ajax/dialog" (because it is in the ajax sub-folder). This name is problematic, as it is not valid to define an XML element with an element name `<t:ajax/dialog>`. Instead, just replace the slashes with periods: `<t:ajax.dialog>`. Library namespaces (described in the next section) are a preferred way of handling components in sub-folders.

When using a component library, the library will map its components to a *virtual sub-folder* of the application. The same naming mechanism works whether its is a real sub-folder or a component library sub-folder.

Library Namespaces

If you are using many components from a common Tapestry component library, you can use a special namespace to simplify references to those components.

The special namespace URI `tapestry-library:path` can be defined; the path is a prefix used in conjunction with component element names.

Borrowing from the above example, all of the following are equivalent:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_3.xsd" xmlns:a="tapestry-library:ajax">

  <t:ajax.dialog/>

  <span t:type="ajax/dialog"/>

  <a:dialog/>

  . . .
```

In other words, the virtual folder, `ajax`, defined in the namespace URI takes the place of the path prefixes `ajax.` seen in the first element, or `ajax/` seen in the second. As far as Tapestry is concerned, they are all equivalent.

Invisible Instrumentation

A favorite feature of Tapestry is *invisible instrumentation*, the ability to mark ordinary HTML elements as components. Invisible instrumentation leads to more concise templates that are also more readable.

Invisible instrumentation involves using *namespaced* id or type attributes to mark an ordinary (X)HTML element as a component. For example:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_3.xsd">
<p>
    Merry Christmas:
    <span t:type="Count" end="3">
        Ho!
    </span>
</p>
```

The t:type attribute above marks the span element as a component of type Count. When rendered, the span element will be *replaced* by the output of the Count component.

The id, type and mixins attributes must be placed in the Tapestry namespace (almost always as t:id, t:type, and t:mixins). Any additional attributes may be in either the Tapestry namespace or the default namespace. Placing an attribute in the Tapestry namespace is useful when the attribute is not defined for the element being instrumented.

An invisibly-instrumented component must still have a type, identified in one of two ways:

- via the t:type attribute in the containing template, as in the above example, or
- within the containing component's Java class using the @Component annotation (and using the t:id attribute on the element in the template). The Component annotation is attached to a field; the type of the component is determined by either the type of the field or the type attribute of the Component annotation.

In *most* cases, it is merely an aesthetic choice whether to use invisible instrumentation in your templates. However, in a very few cases the behavior of the component is influenced by your choice. For example, when your template includes the Loop component using the invisible instrumentation approach, the original tag (and its informal parameters) will render repeatedly around the body of the component. Thus, for example:

```
  <table>
    <tr t:type="loop" source="items" value="item" class="prop:rowClass">
      <td>${item.id}</td>
      <td>${item.name}</td>
      <td>${item.quantity}</td>
    </tr>
  </tabel>
```

Here, the loop component "merges into" the <tr> element. It will render out a <tr> for each `item` in the `items` list, with each <tr> including three <td> elements. It will also write a dynamic "class" attribute into each <tr>.

Parameter Namespaces

Main Article: Component Parameters

Parameter namespaces (introduced in Tapestry 5.1) are a concise way of passing parameter blocks to components.

You must define a special namespace, usually with the prefix "p":

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_3.xsd" xmlns:p="tapestry:parameter">
  . . .
```

With the "tapestry:parameter" namespace defined, you can pass block using the "p:" prefix and an element name that matches the parameter name:

```
<t:if test="loggedIn">
  Hello, ${userName}!
  <p:else>
    Click <a t:type="actionlink" t:id="login">here</a> to log in.
  </p:else>
</t:if>
```

This example passes a block of the template (containing the ActionLink component and some text) to the If component as parameter `else`. In the If component's reference page you'll see that `else` is parameter of type Block.

Name-spaced parameter elements are not allowed to have any attributes. The element name itself is used to identify the parameter of the component to bind.

Whitespace in Templates

Tapestry strips out unnecessary whitespace from templates as they are parsed. Inside any block of text, repeated whitespace is reduced to a single space character. Blocks of text that are entirely whitespace, such a line break and whitespace between two tags, is eliminated entirely.

If you do a view source on the rendered output, you'll see that the bulk of the rendered page is one long unbroken line.

This approach has certain efficiency advantages on both the server (less processing to render the page) and on the client (fewer characters to parse). Tools such as Firefox Developer Tools and Chrome Developer Tools are useful for allowing you to view the rendered HTML on the client properly.

In rare cases, the whitespace in a template *is* significant. Perhaps you are creating a <pre> (preformatted) block of text, or the whitespace interacts with your stylesheet to some desired effect.

You may use the standard XML attribute `xml:space` to indicate to Tapestry whether whitespace should be compressed (`xml:space="default"`) or preserved (`xml:space="preserve"`). Such attributes are stripped out by the template parser; they do not appear in the rendered output.

The xml: namespace prefix is built into all XML documents, there is no special configuration (as there is with the Tapestry namespace).

For example:

```
  <ul class="navmenu" xml:space="preserve">
    <li t:type="loop" t:source="pages" t:value="var:page">
      <t:pagelink page="var:page">${var:page}</t:pagelink>
    </li>
  </ul>
```

This will preserve the whitespace between the <ul> and <li> elements, and between the (rendered) elements and the nested <a> elements. For example, the output may look something like:

```
  <ul>
    <li>
      <a href="showcart>ShowCart</a>
    </li>
    <li>
      <a href="viewaccount">ViewAccount</a>
    </li>
  </ul>
```

With normal whitespace compression, you would see the following rendered output:

```
  <ul><li><a href="showcart">ShowCart</a></li><li><a href="viewaccount">ViewAccount</li></ul>
```

You can even put further `xml:space` attributes inside nested elements to fine-tune the control over what whitespace is preserved and what is compressed.

Template Inheritance

If a component does not have a template, but extends from a component class that does have a template, then the parent class' template will be used by the child component.

This allows a component to extend from a base class but not have to duplicate the base class' template.

Tapestry 5.1 adds a significant new feature: template inheritance with *extension points*. Previously, a component which extended another component had to inherit the parent component's entire template, or copy-and-paste the template.

Parent template can now mark replaceable sections as <t:extension-point>s, and sub-components can extend the parent template and <t:replace> those sections.

This can work across multiple levels of inheritance.

Overuse of this feature is *not recommended*: in general use of composition, rather than inheritance, will be easier to understand and maintain. There are certain specific cases where overrides will allow a for much wider and easier reuse of a component, but the component needs to be carefully designed to support this kind of inheritance properly.

<t:extension-point>

Marks a point in a template that may be replaced. A unique id (case insensitive) is used in the template and its sub-templates to link extension points to possible overrides.

```
  <t:extension-point id="title">
    <h1>${defaultTitle}</h1>
  </t:extension-point>
```

<t:extend>

Root element of a child template that extends from its parent template. The <t:extend> attribute may only appear as the root element and may only contain <t:replace> elements.

<t:replace>

Replaces an extension point from a parent template. <t:replace> may only appear as the immediate child of a root <t:extend> element.

```
<t:extend xmlns:t="http://tapestry.apache.org/schema/tapestry_5_3.xsd">
  <t:replace id="title">
    <h1><img src="${context:images/icon.jpg}"/>
    Customer Service</h1>
  </t:replace>
</t:extend>
```