Templating and Markup FAQ

Templating and Markup

Main Article: Component Templates

Contents

Why do I get a SAXParseException when I use an HTML entity, such as ` ` in my template?

Tapestry uses a standard SAX parser to read your templates. This means that your templates must be *well formed*: open and close tags must balance, attribute values must be quoted, and entities must be declared. The easiest way to accomplish this is to add a DOCTYPE to your the top of your template:

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

Part of the DOCTYPE is the declaration of entities such as ` `.

Alternately, you can simply use the numeric version: ` ` This is the exact same character and will render identically in the browser.

Starting in release 5.3, Tapestry introduces an XHTML doctype when no doctype is present; this means that common HTML entities will work correctly.

Why do some images in my page show up as broken links?

You have to be careful when using relative URLs inside page templates; the base URL may not always be what you expect. For example, inside your `ViewUser.tml` file, you may have:

```
  <img class="icon" src="icons/admin.png"/>${user.name} has Administrative access
```

This makes sense; ViewUser.tml is in the web context, as is the icons folder. The default URL for this page will be /viewuser (assuming that ViewUser class is in the *root-package*.pages package).

However, the ViewUser page might use a page activation context to identify which user is to be displayed:

```
public class ViewUser

  @Property
  @PageActivationContext
  private User user;

  . . .
```

With a page activation context, the URL for the page will incorporate the ID of the User object, something like `/viewuser/37371`. This is why the relative URL to the `admin.png` image is broken: the base path is relative to the page's URL, not to the page template. (In fact, the page template may not even be in the web context, it may be stored on the classpath, as component templates are.)

One solution would be to predict what the page URL will be, and adjust the path for that:

```
  <img class="icon" src="../icons/admin.png"/>${user.name} has Administrative access
```

But this has its own problems; the page activation context may vary in length at different times, or the template in question may be a component used across many different pages, making it difficult to predict what the correct relative URL would be.

The *best* solution for this situation, one that will be sure to work in all pages and all components, is to make use of the `context:` binding prefix:

```
  <img class="icon" src="${context:icons/admin.png}"/>${user.name} has Administrative access
```

The src attribute of the <img> tag will now be bound to a dynamically computed value: the location of the image file relative to the web application context. This is especially important for components that may be used on different pages.

What's the difference between `id` and `t:id`?

You might occasionally see something like the following in a template:

```
<t:zone id="status" t:id="statusZone">
```

Why two ids? Why are they different?

The `t:id` attribute is the Tapestry component id. This id is unique within its immediate container. This is the id you might use to inject the component into your page class:

```
  @InjectComponent
  private Zone statusZone;
```

The other id is the client id, a unique id for the rendered element within the client-side DOM. JavaScript that needs to access the element uses this id. For example:

```
  $('status').hide();
```

In many components, the `id` attribute is an informal parameter; a value from the template that is blindly echoed into the output document. In other cases, the component itself has an `id` attribute. Often, in the latter case, the Tapestry component id is the *default* value for the client id.

Why do my images and stylesheets end up with a weird URLs like `/assets/meta/zeea17aee26bc0cae/layout/layout.css`?

Tapestry doesn't rely on the servlet container to serve up your static assets (images, stylesheets, flash movies, etc.). Instead, Tapestry processes the requests itself, streaming assets to the browser.

Asset content will be GZIP compressed (if the client supports compression, and the content is compressible). In addition, Tapestry will set a far-future expires header on the content. This means that the browser will not ask for the file again, greatly reducing network traffic.

The weird hex string is a *fingerprint*; it is a hash code computed from the actual content of the asset. If the asset ever changes, it will have a new fingerprint, and so will be a new path and a new (immutable) resource. This approach, combined with a far-future expires header also provided by Tapestry, ensures that clients aggressively cache assets as they navigate your site, or even between visits.

How do I add a CSS class to a Tapestry component?

As they say, "just do it". The majority of Tapestry components support *informal parameters*, meaning that any extra attributes in the element (in the template) will be rendered out as additional attributes. So, you can apply a CSS class or style quite easily:

```
  <t:textfield t:id="username" class="big-green"/>
```

You can even use template expansions inside the attribute value:

```
  <t:textfield t:id="username" class="${usernameClass}"/>
```

and

```
  public String getUsernameClass()
  {
    return isUrgent() ? "urgent" : null;
  }
```

When an informal parameter is bound to null, then the attribute is not written out at all.

You can verify which components support informal parameters by checking the component reference, or looking for the @SupportsInformalParameters annotation in the components' source file.