Page And Component Classes FAQ

Page And Component Classes

Main article: Component Classes

Contents

What's the difference between a page and a component?

There's very little difference between the two. Pages classes must be in the *root-package*.`pages` package; components must be in the *root-package*.`components`. Pages may provide event handlers for certain page-specific events (such as activate and passivate). Components may have parameters.

Other than that, they are more equal than they are different. They may have templates or may render themselves in code (pages usually have a template, components are more likely to render only in code).

The major difference is that Tapestry page templates may be stored in the web context directory, as if they were static files (they can't be accessed from the client however; a specific rule prevents access to files with the `.tml` extension).

It is possible that this feature may be removed in a later release. It is preferred that page templates be stored on the classpath, like component templates.

How do I store my page classes in a different package?

Tapestry is very rigid here; you can't. Page classes must go in *root-package*.`pages`, component classes in *root-package*.`components`, etc.

You are allowed to create sub-packages, to help organize your code better and more logically. For example, you might have *root-package*.`pages.account.ViewAccount`, which would have the page name "account/viewaccount". (Tapestry would also create an alias "account/view", by stripping off the redundant "account" suffix. Either name is equally valid in your code, and Tapestry will use the shorter name, "account/view" in URLs.)

In addition, it is possible to define additional root packages for the application:

```
public static void contributeComponentClassResolver(Configuration<LibraryMapping> configuration) {
       configuration.add(new LibraryMapping("", "com.example.app.tasks"));
       configuration.add(new LibraryMapping("", "com.example.app.chat"));
}
```

LibraryMappings are used to resolve a library prefix to one or more package names. The empty string represents the application itself; the above example adds two additional root packages; you might see additional pages under `com.example.app.tasks.pages`, for example.

Tapestry doesn't check for name collisions, and the order the packages are searched for pages and components is not defined. In general, if you can get by with a single root package for your application, that is better.

Why do my instance variables have to be private?

In Tapestry 5.3.1 and earlier all instance variables must be private. Starting in version 5.3.2 instance variables can also be protected or package private (that is, not public), or they can even be public if `final` or annotated with the deprecated @Retain.

Tapestry does a large amount of transformation to your simple POJO classes as it loads them into memory. In many cases, it must locate every read or write of an instance variable and change its behavior; for example, reading a field that is a component parameter will cause a property of the containing page or component to be read.

Restricting the scope of fields allows Tapestry to do the necessary processing one class at a time, as needed, at runtime. More complex Aspect Orient Programming systems such as AspectJ can perform similar transformations (and much more complex ones), but they require a dedicated build step (or the introduction of a JVM agent).

Why don't my informal parameters show up in the rendered markup?

Getting informal parameters to work is in two steps. First, you must make a call to the `ComponentResources.renderInformalParameters()` method, but just as importantly, you must tell Tapestry that you want the component to support informal parameters, using the `SupportsInformalParameters` annotation. Here's a hypothetical component that displays an image based on the value of a `Image` object (presumably, a database entity):

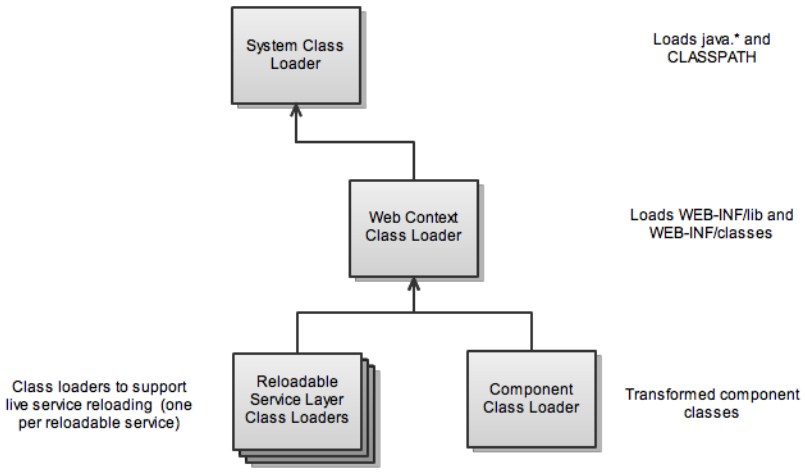
```
@SupportsInformalParameters
public class DBImage
{
  @Parameter(required=true)
  private Image image;

  @Inject
  private ComponentResources resources;

  boolean beginRender(MarkupWriter writer)
  {
    writer.element("img", "src", image.toClientURL(), "class", "db-image");

    resources.renderInformalParameters(writer);

    writer.end();

    return false;
  }
}
```

Why do I get java.lang.LinkageError when I invoke public methods of my page classes?

In Tapestry, there are always *two* versions of page (or component) classes. The first version is the version loaded by standard class loader: the simple POJO version that you wrote.

The second version is much more complicated; it's the transformed version of your code, with lots of extra hooks and changes to allow the class to operate inside Tapestry. This includes implementing new interfaces and methods, adding new constructors, and changing access to existing fields and methods.

Although these two classes have the same fully qualified class name, they are distinct classes because they are loaded by different class loaders.



In a Tapestry application, most application classes are loaded from the middle class loader. Additional class loaders are used to support live service reloading, and live component reloading (along with component class transformation).

When a page or component is passed as a parameter to a service, a failure occurs (how it is reported varies in different JDK releases) because of the class mismatch.

The solution is to define an interface with the methods that the service will invoke on the page or component instance. The service will expect an object implementing the interface (and doesn't care what class loader loaded the implementing class).

Just be sure to put the interface class in a non-controlled package, such as your application's *root-package* (and **not** *root-package*.`pages`).

Which is better, using magic method names (i.e., `beginRender()`) or annotations (i.e. `BeginRender`)?

There is no single best way; this is where your taste may vary. Historically, the annotations came first, and the method naming conventions came later.

The advantage of using the method naming conventions is that the method names are more concise, which fewer characters to type, and fewer classes to import.

The main disadvantage of the method naming conventions is that the method names are not meaningful. `onSuccessFromLoginForm()` is a less meaningful name than `storeUserCredentialsAndReturnToProductsPage()`, for example.

The second disadvantage is you are more susceptible to off-by-a-character errors. For example, `onSucessFromLoginForm()` will *never* be called because the event name is misspelled; this would not happen using the annotation approach:

```
@OnEvent(value=EventConstants.SUCCESS, component="loginForm")
Object storeUserCredentialsAndReturnToProductsPage()
{
  . . .
}
```

The compiler will catch a misspelling of the constant SUCCESS. Likewise, local constants can be defined for key components, such as "loginForm".

Ultimately, it's developer choice. HLS prefers the method naming conventions in nearly all cases, especially prototypes and demos, but can see that in some projects and some teams, an annotation-only approach is best.

Why do I have to inject a page? Why can't I just create one using new?

Tapestry tranforms your class at runtime. It tends to build a large constructor for the class instance. Further, an instance of the class is useless by itself, it must be wired together with its template and its sub-components.

On top of that, Tapestry keeps just once instance of each page in memory (since 5.2). It reworks the bytecode of the components so that a single instance can be shared across multiple request handling threads.

Templating and Markup FAQ | Frequently Asked Questions | Forms and Form Components FAQ