

Component Classes

A **component class** is the class associated with a page, component or mixin in your Tapestry web application. Classes for pages, components and mixins are all created in an identical way. They are pure POJOs (Plain Old Java Objects), typically with annotations and conventionally named methods. They are not *abstract*, nor do they need to extend base classes or implement interfaces.

In most cases, each component class will have a corresponding [component template](#). However, it is also possible for a component class to emit all of its markup itself, without using a template.

For Tapestry 4 Users: Component classes in Tapestry 5 are much easier than in Tapestry 4. There are no base classes to extend from, the classes are concrete (not abstract), and there's no XML file. There is still a bit of configuration in the form of Java annotations, but those now go directly onto fields of your class, rather than on abstract getters and setters.

Related Articles

- [Component Libraries](#)
- [Component Reference](#)
- [Page And Component Classes FAQ](#)
- [Component Templates](#)
- [Component Cheat Sheet](#)
- [Component Parameters](#)

Creating a Trivial Component

Creating a page or component in Tapestry 5 is a breeze. There are only a few constraints:

- There must be a public Java class.
- The class must be in the correct package (see below).
- The class must have a public, no-arguments constructor. (The default one provided by the compiler is fine.)

Here's a minimal component that outputs a fixed message, using a [template](#) with a matching file name:

HelloWorld.java

```
package org.example.myapp.components;
public class HelloWorld
{
}
```

HelloWorld.tml

```
<html>
  Bonjour from HelloWorld component.
</html>
```

In the example above, the HelloWorld class contains no code at all (except what it inherits from the Object class and what Tapestry adds invisibly).

And here's a component that does the same thing, but without needing a template:

HelloWorld.java – without a template

```
package org.example.myapp.components;

import org.apache.tapestry5.MarkupWriter;
import org.apache.tapestry5.annotations.BeginRender;

public class HelloWorld
{
    @BeginRender
    void renderMessage(MarkupWriter writer)
    {
        writer.write("Bonjour from HelloWorld component.");
    }
}
```

In this example, just like the first one, the component's only job is to write out a fixed message. The [@BeginRender](#) annotation is a type of [render phase annotation](#)

, a method annotation that instructs Tapestry when and under what circumstances to invoke methods of your class.

These methods are not necessarily public; they can have any access level you like (unlike in Tapestry 4). By convention they usually have package-private access level (the default).

Component Packages

Component classes must exist within an appropriate package (this is necessary for runtime code transformation and class reloading to operate).

These packages exist under the application's root package, as follows:

- For pages, place classes in **root.pages**. Page names are mapped to classes within this package.
- For mixins, place classes in **root.mixins**. Mixin types are mapped to classes within this package.
- For other components, place classes in **root.components**. Component types are mapped to classes within this package.

In addition, it is common for an application to have base classes, often *abstract* base classes, that should not be directly referenced. These should *not* go in the **pages**, **components** or **mixins** packages, because they then look like valid pages, components or mixins. Instead, use the **root.base** package to store such base classes.



Only component classes should go in any of these controlled packages; classes representing data, or interfaces, or anything that isn't precisely a component class, must go elsewhere. Any top-level class in any of the controlled packages will be transformed at runtime. The only exception is inner classes (anonymous or not), which are loaded by the same class loader as the component class loader, but not transformed as components.

Sub-Folders / Sub-Packages

Classes do not have to go directly inside the package (pages, components, mixins, etc.). It is valid to create a sub-package to store some of the classes. The sub-package name becomes part of the page name or component type. Thus you might define a page component `com.example.myapp.pages.admin.CreateUser` and the logical page name (which often shows up inside URLs) will be **admin/CreateUser**.

Tapestry performs some simple optimizations of the logical page name (or component type, or mixin type). It checks to see if the package name is either a prefix or a suffix of the unqualified class name (case insensitively, of course) and removes the prefix or suffix if so. The net result is that a class name such as `com.example.myapp.pages.user.EditUser` will have a page name of `user/Edit` (instead of `user/EditUser`). The goal here is to provide shorter, more natural URLs.

Index Pages

One special simplification exists for Index pages: if the logical page name is Index after removing the package name from the unqualified class name, it will map to the root of that folder. A class such as `com.example.myapp.pages.user.IndexUser` or `com.example.myapp.pages.user.UserIndex` will have a page name of `user/`.

In previous versions of Tapestry there was also the concept of a start page configured with the `tapestry.start-page-name` configuration symbol (defaults to "start"). If a page with a name as configured with that symbol exists at the root level, this page is used as the root URL. This has precedence over an existing Index page. If for example you have a page class `com.example.myapp.pages.Start` it will map to `/`.



Use of start-pages is discouraged and support for it will eventually be removed. Use an Index page instead.

Pages vs. Components

The distinction between pages and component is very, very small. The primary difference is the package name: **root.pages**.*PageName* for pages, and **root.components**.*ComponentType* for components. Conceptually, page components are simply the *root component* of a page's component tree.

For Tapestry 4 users: there was a much greater distinction in Tapestry 4 between pages and components, which showed up as separate interfaces and a hierarchy of abstract implementations to extend your classes from.

Class Transformation

Tapestry uses your class as a starting point. It *transforms* your class at runtime. This is necessary for a number of reasons, including to address how Tapestry shares pages between requests.

For the most part, these transformations are both sensible and invisible. In a few limited cases, they comprise a marginally [leaky abstraction](#) – for instance, the scope restrictions on instance variables described below – but the programming model in general supports a very high level of developer productivity.

Because transformation doesn't occur until *runtime*, the build stage of your application is not affected by the fact that you are creating a Tapestry application. Further, your classes are absolutely simple POJOs during unit testing.

Live Class Reloading

Main Article: [Class Reloading](#)

Component classes are monitored for changes by the framework. [Classes are reloaded when changed](#). This allows you to build your application with a speed approaching that of a scripting environment, without sacrificing any of the power of the Java platform.

And it's fast! You won't even notice that this magic class reloading has occurred.

The net result: super productivity — change your class, see the change instantly. This is designed to be a blend of the best of scripting environments (such as Python or Ruby) with all the speed and power of Java backing it up.

However, class reloading *only* applies to component classes (pages, components and mixins) and, starting in 5.2, Tapestry IOC-based service implementations (with some restrictions). Other classes, such as service interfaces, entity/model classes, and other data objects, are loaded by the normal class loader and not subject to live class reloading.

Instance Variables

Tapestry components may have instance variables (unlike Tapestry 4, where you had to use *abstract properties*).

Since release 5.3.2, instance variables may be protected, or package private (that is, no access modifier). Under specific circumstances they may even be public (public fields must either be final, or have the [@Retain](#) annotation).

Be aware that you will need to either provide getter and setter methods to access your classes' instance variables, or else annotate the fields with [@Property](#).

Transient Instance Variables

Unless an instance variable is decorated with an annotation, it will be a *transient* instance variable. This means that its value resets to its default value at the end of each request (when the [page is detached from the request](#)).



About initialization

Never initialize an instance field to a *mutable* object at the point of declaration. If this is done, the instance created from that initializer becomes the default value for that field and is reused inside the component on every request. This could cause state to inadvertently be shared between different sessions in an application.

Deprecated since 5.2

For Tapestry 5.1 and earlier, in the rare event that you have a variable that can keep its value between requests and you would like to defeat that reset logic, then you can add a [@Retain](#) annotation to the field. You should take care that no client-specific data is stored into such a field, since on a later request the same page *instance* may be used for a different user. Likewise, on a later request for the *same* client, a *different* page instance may be used.

Use [persistent fields](#) to hold client-specific information from one request to the next.

Further, final fields are (in fact) final, and will not be reset between requests.

Constructors

Tapestry will instantiate your class using the default, no arguments constructor. Other constructors will be ignored.

Injection

Main Article: [Injection](#)

Injection of dependencies occurs at the field level, via additional annotations. At runtime, fields that contain injections become read-only.

```

@Inject // inject a resource
private ComponentResources componentResources;

@Inject // inject a block
private Block foo;

@Inject // inject an asset
@Path("context:images/top_banner.png")
private Asset banner;

@Inject // inject a service
private AjaxResponseRenderer ajaxResponseRenderer;

```

Parameters

Main Article: [Component Parameters](#)

Component parameters are private fields of your component class annotated with `@Parameter`. Component parameters represent a two-way binding of a field of your component and a property or resource of its containing component or page.

Persistent Fields

Main Article: [Persistent Page Data](#)

Most fields in component classes are automatically cleared at the end of each request. However, fields may be annotated so that they retain their value across requests, using the `@Persist` annotation.

Embedded Components

Components often contain other components. Components inside another component's template are called *embedded components*. The containing component's [template](#) will contain special elements, in the Tapestry namespace, identifying where the embedded components go.

You can define the type of component inside template, or you can create an instance variable for the component and use the `@Component` annotation to define the component type and parameters.

Example:

```

package org.example.app.pages;

import org.apache.tapestry5.annotations.Component;
import org.apache.tapestry5.annotations.Property;
import org.example.app.components.Count;

public class Countdown
{
    @Component(parameters =
    { "start=5", "end=1", "value=countValue" })
    private Count count;

    @Property
    private int countValue;
}

```

The above defines a component whose embedded id is "count" (this id is derived from the name of the field and an element with that id must be present in the corresponding template, otherwise an error is displayed (see below)). The type of the component is `org.example.app.components.Count`. The start and end parameters of the Count component are bound to literal values, and the value parameter of the Count component is bound to the `countValue` property of the Countdown component.


Technically, the start and end parameters should be bound to properties, just like the value parameter. However, certain literal values, such as the numeric literals in the example, are accepted by the `prop:` binding prefix even though they are not actually properties (this is largely as a convenience to the application developer). We could also use the `literal:` prefix, `"start=literal:5"`, which accomplishes largely the same thing.

You may specify additional parameters inside the component template, but parameters in the component class take precedence.

TODO: May want a more complex check; what if user uses `prop:` in the template and there's a conflict?

You may override the default component id (as derived from the field name) using the `id()` attribute of the `Component` annotation.

If you define a component in the component class, and there is no corresponding element in the template, Tapestry will log an error. In the example above that would be the case if the template for the Countdown page didn't contain an element with `<t:count t:id="count">`.

 [Annotations](#)

 [User Guide](#)

[Component Templates](#) 