}

Component Classes :

A **component class** is the class associated with a page, component or mixin in your Tapestry web application. They are pure POJOs (Plain Old Java Objects), typically with annotations and conventionally named methods.

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>
   HelloWorld component.
</html>
```

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.

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.

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.

Instance Variables

Tapestry components may have instance variables
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.

Constructors

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

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;
```

Persistent Fields

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 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

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">.

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.

A template for a page

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

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.

Tapestry Elements

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

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>

Expansions

Another option when rendering output is the use of *expansions*. Expansions are special strings that may be embedded in template bodies

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.

Embedded Components

An embedded component is identified within the template as an element in the t: namespace.
Example:
<t:actionlink t:id="clear">Remove All</t:actionlink>.

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

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.

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.

t 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="user">
            Welcome back, ${user.firstName}
            <p:else>
                <t:pagelink page="login">Login</t:pagelink> /
                <t:pagelink page="register">Register</t:pagelink>
            </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.

Component Parameters:

Component parameters are the primary means for a component instance and its container to communicate with each other. Parameters are used to *configure* component instances.

In the following example, page is a parameter of the pagelink component. The page parameter tells the pagelink component which page to go to when the user clicks on the rendered hyperlink:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
    <t:pagelink page="Index">Go Home</t:pagelink>
</html>
```

A component may have any number of parameters. Each parameter has a specific name, a specific Java type (which may be a primitive value), and may be *optional* or *required*.
Within a component class, parameters are declared by using the @Parameter annotation on a private field, as we'll see below.

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.

<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>
```

Property Expressions :

Tapestry uses **property expressions** to move data between components. Property expressions are the basis of the component parameters and template expansions.

user?.name
Calls getUser() and, if the result is not null, calls getName() on the result

Component Parameters :

Component parameters are the primary means for a component instance and its container to communicate with each other. Parameters are used to *configure* component instances.

In the following example, page is a parameter of the pagelink component. The page parameter tells the pagelink component which page to go to when the user clicks on the rendered hyperlink:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
   <t:pagelink page="Index">Go Home</t:pagelink>
</html>
```

Parameter Bindings

In Tapestry, a parameter is not a slot into which data is pushed: it is a *connection* between a field of the component (marked with the @Parameter annotation) and a property or resource of the component's container. (Components can be nested, so the container can be either the page or another component.)

The connection between a component and a property (or resource) of its container is called a *binding*. The binding is two-way: the component can read the bound property by reading its parameter field. Likewise, a component that updates its parameter field will update the bound property.

This is important in a lot of cases; for example a TextField component can read *and update* the property bound to its value parameter. It reads the value when rendering, but updates the value when the form is submitted.
The component listed below is a looping component; it renders its body a number of times, defined by its start and end parameters (which set the boundaries of the loop). The component can update a result parameter bound to a property of its container; it will automatically count up or down depending on whether start or end is larger.

```
package org.example.app.components;

import org.apache.tapestry5.annotations.AfterRender;
import org.apache.tapestry5.annotations.Parameter;
import org.apache.tapestry5.annotations.SetupRender;

public class Count
{
   @Parameter (value="1")
   private int start;

   @Parameter(required = true)
   private int end;

   @Parameter
   private int result;

   private boolean increment;

   @SetupRender
   void initializeValues()
   {
      result = start;
      increment = start < end;
   }

   @AfterRender
   boolean next()
   {
      if (increment)
      {
         int newResult = value + 1;

         if (newResult <= end)
         {
           result = newResult;
            return false;
         }
      }
      else
      {
         int newResult= value - 1;
         if (newResult>= end)
         {
            result = newResult;
            return false;
         }
      }
      return true;
   }
}
```

The name of the parameter is the same as field name (except with leading "_" and "$" characters, if any, removed). Here, the parameter names are "start", "end" and "result".
The component above can be referenced in another component or page template, and its parameters *bound*:

```
<html t:type="layout"
xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
    <p> Merry Christmas: <t:count end="3"> Ho! </t:count>
    </p>
</html>
```

The end attribute is used to *bind* the end parameter of the Count component. Here, it is being bound to the string value "3", which is automatically coerced by Tapestry into the int value, 3.

Publishing Parameters

Often when creating new components from existing components, you want to expose some of the functionality of the embedded component, in the form of exposing parameters of the embedded components as parameters of the outer component.
In Tapestry 5.0, you would define a parameter of the outer component, and use the "inherit:" binding prefix to connect the inner component's parameter to the outer component's parameter.

This is somewhat clumsy, as it involves creating an otherwise unused field just for the parameter; in practice it also leads to duplication of the documentation of the parameter.
In Tapestry 5.1 and later, you may use the publishParameters attribute of the @Component annotation. List one or more parameters separated by commas: those parameters of the inner/embedded component become parameters of the outer component. You should **not** define a parameter field in the outer component.

ContainerComponent.tml

```
<t:container xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
<t:pageLink t:id="link">Page Link</t:pageLink>
</t:container>
```

ContainerComponent.java

```
public class ContainerComponent{
   @Component(id="link", publishParameters="page")
   private PageLink link;
}
```

Index.tml

```
<t:ContainerComponent t:id="Container" t:page="About" />
```

Layout Component:

The Layout component is a component *that you create* to provide common elements across all of your pages.

Layout.tml (a template for a Layout component)

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_4.xsd">
   <head>
      <title>My Nifty Web Application</title>
   </head>
   <body>
      <div class="nav-top">
         Nifty Web Application
      </div>

      <t:body/>

      <div class="nav-bottom">
         (C) 2012 NiftyWebCo, Inc.
      </div>
```

```
  </body>
</html>
```

Welcome.tml (the template for a page)

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

  <h1>Welcome to the Nifty Web Application!</h1>

  <p>
      Would you like to <t:pagelink page="login">Log In</t:pagelink>?
  </p>
</html>
```

Note the "t:type="layout" part. That says, in effect, "wrap the layout component around my *content*".
The magic is in the <t:body/> element of the layout template; this will be replaced by each page's *content*, whatever that is.

Request/Response Processing:

Page Life Cycle:

In Tapestry, you are free to develop your presentation objects, page and components classes, as ordinary objects, complete with instance variables and so forth.

This is somewhat revolutionary in terms of web development in Java. By comparison, using traditional servlets, or Struts, your presentation objects (Servlets, or Struts Actions, or the equivalent in other frameworks) are *stateless singletons*. That is, a *single* instance is created, and all incoming requests are threaded through that single instance. Because multiple requests are handled by many different threads, this means that the singleton's instance variables are useless ... any value written into an instance variable would immediately be overwritten by a different thread. Thus, it is necessary to use the Servlet API's HttpServletRequest object to store per-request data, and the HttpSession object to store data between requests.

Tapestry takes a very different approach.
In Tapestry, each page is a singleton, but with a *per thread* map of field names & values that Tapestry invisibly manages for you.

With this approach, all the difficult, ugly issues related to multi-threading go by the wayside. Instead, familiar, simple coding practices (using ordinary methods and fields) can be used.

Tapestry 5.0 and 5.1 used page pooling, rather than a singleton page with a per-thread map, to achieve the same effect.

The page life cycle is quite simple:

1. When first needed, a page is loaded. Loading a page involves instantiating the components of the page and connecting them together.
2. Once a page is loaded, it is *attached* to the current request. Remember that there will be many threads, each handling its own request to the same page.
3. At the end of a request, after a response has been sent to the client, the page is *detached* from the request. This is a chance to perform any cleanup needed for the page.

Page Life Cycle Methods

There are rare occasions where it is useful for a component to perform some operations, usually some kind of initialization or caching, based on the life cycle of the page.
As with component rendering, you have the ability to make your components "aware" of these events by telling Tapestry what methods to invoke for each.
Page life cycle methods should take no parameters and return void.
You have the choice of attaching an annotation to a method, or simply using the method naming conventions:

Annotation	Method Name	When Called
@PageLoaded	pageLoaded()	After the page is fully loaded
@PageAttached	pageAttached()	After the page is attached to the request.
@PageReset	pageReset()	After the page is *activated*, except when requesting the same page
@PageDetached	pageDetached()	AFter the page is detached from the request.

The @PageReset life cycle (only for Tapestry 5.2 and later) is invoked on a page render request when the page is linked to from some *other* page of the application (but *not* on a link to the same page), or upon a reload of the page in the browser. This is to allow the page to reset its state, if any, when a user returns to the page from some other part of the application.

In Tapestry 5.0 and 5.1, a page pool is used to store page instances. The pool is "keyed" on the name of the page (such as "start") and the *locale* for the page (such as "en" or "fr").
Within each key, Tapestry tracks the number of page instances that have been created, as well as the number that are in use (currently attached to a request).
When a page is first accessed in a request, it is taken from the pool. Tapestry has some configuration values that control the details of how and when page instances are created.

- If a free page instance is available, the page is marked in use and attached to the request.
- If there are fewer page instances than the *soft limit*, then a new page instance is simply created and attached to the request.
- If the soft limit has been reached, Tapestry will wait for a short period of time for a page instance to become available before creating a new page instance.
- If the hard limit has been reached, Tapestry will throw an exception rather than create a new page instance.
- Otherwise, Tapestry will create a new page instance.
  Thus a busy application will initially create pages up-to the soft limit (which defaults to five page instances). If the application continues to be pounded with requests, it will slow its request processing, using the soft wait time in an attempt to reuse an existing page instance.

Request Processing :

Request Processing involves a sequence of steps that Tapestry performs when every HTTP request comes in. You *don't need* to know these steps to use Tapestry productively, but understanding the request processing pipeline is helpful if you want to understand Tapestry deeply.
Much of the early stages of processing are in the form of extensible pipelines.

Tapestry Filter

All incoming requests originate with the TapestryFilter, which is a servlet filter configured inside your application's web.xml.
The TapestryFilter is responsible for a number of startup and initialization functions.
When it receives a request, the TapestryFilter obtains the HttpServletRequestHandler service, and invokes its service() method.

Component Rendering:
Rendering of components in Tapestry 5 is based on a *state machine* and a *queue* (instead of the tail recursion used in Tapestry 4). This breaks the rendering process up into tiny pieces that can easily be implemented or overridden.

All Render phase methods are *optional*; a default behavior is associated with each phase.

Annotation	Method Name	When Called
@SetupRender	setupRender()	When initial setup actions, if any, are needed
@BeginRender	beginRender()	When Tapestry is ready for the component's start tag, if any, to be rendered
@BeforeRenderTemplate	beforeRenderTemplate()	Before Tapestry renders the component's template, if any
@BeforeRenderBody	beforeRenderBody()	Before Tapestry renders the body of the component, if any
@AfterRenderBody	afterRenderBody()	After Tapestry renders the body of the component, if any, but before the rest of the component's template is rendered
@AfterRenderTemplate	afterRenderTemplate()	After Tapestry finishes rendering the component's template, if any
@AfterRender	afterRender()	After Tapestry has finished rendering both the template and body of the component

@CleanupRender	cleanupRender()	When final cleanup actions, if any, are needed

DOM (**Document Object Model**):

Once the Document object is created, you don't directly create new DOM objects; instead, each DOM object includes methods that create new sub-objects. This primarily applies to the Element class, which can be a container of text, comments and other elements.

Response Compression:

Starting in Tapestry 5.1, the framework automatically GZIP **compresses** content streamed to the client. This can significantly reduce the amount of network traffic for a Tapestry application, at the cost of extra processing time on the server to compress the response stream.

This directly applies to both rendered pages and streamed assets from the CLASSPATH. Context assets will also be compressed ... but this requires referencing such assets using the "context:" binding prefix, so that generated URL is handled by Tapestry and not the servlet container.

Small streams generally do not benefit from being compressed; there is overhead when using compression, not just the CPU time to compress the bytes, but a lot of overhead. For small responses, Tapestry does not attempt to compress the output stream.
The configuration symbol tapestry.min-gzip-size allows the cutoff to be set; it defaults to 100 bytes.

In addition, some file types are already compressed and should not be re-compressed (they actually get larger, not smaller!). The service ResponseCompressionAnalyzer's configuration is an unordered collection of content type strings that should *not* be compressed. The default list of content types that are NOT compressed are:

- image/* (image/jpeg, image/png, image/gif, etc) *except* image/svg+xml, which is compressed
- application/x-shockwave-flash
- application/font-woff
- application/x-font-ttf
- application/vnd.ms-fontobj

StreamResponse

When returning a StreamResponse from a component event method, the stream is totally under your control; it will not be compressed. You should use the ResponseCompressionAnalyzer service to determine if the client supports compression, and add a java.util.zip.GZIPOutputStream to your stream stack if compression is desired.

HTTPS:

By default, Tapestry assumes your application will be primarily deployed as a standard web application, using HTTP (not HTTPS) as the primary protocol.

Many applications will need to have some of their pages secured: only accessible via HTTPS. This could be a login page, or a product ordering wizard, or administrative pages.

All that is necessary to mark a page as secure is to add the @Secure annotation to the page class:

```
@Secure
public class ProcessOrder
{
  . . .
}
```

When a page is marked as secure, Tapestry will ensure that access to that page uses HTTPS.
All links to the page will use the "https" protocol.
If an attempt is made to access a secure page using a non-secure request (a normal HTTP request), Tapestry will send an HTTPS redirect to the client.
Links to non-secure pages from a secure page will do the reverse: a complete URL with an "http" protocol will be used. In other words, Tapestry manages the transition from insecure to secure and back again.
Links to other (secure) pages *and to assets* will be based on relative URLs and, therefore, secure.
The rationale behind using secure links to assets from secure pages is that it prevents the client web browser from reporting a mixed security level.

Securing Multiple Pages

Rather than placing an @Secure annotation on individual pages, it is possible to enable https URL redirecting for entire folders of pages. All pages in or beneath the folder will be secured.
This is accomplished by making a contribution to the MetaDataLocator service configuration.
For example, to secure all pages in the "admin" folder:
AppModule.java (partial)

```
public void contributeMetaDataLocator(MappedConfiguration<String,String> configuration)
{
    configuration.add("admin:" + MetaDataConstants.SECURE_PAGE, "true");
}
```

Here "admin" is the folder name, and the colon is a separator between the folder name and the the meta data key. SECURE_PAGE is a public constant for value "tapestry.secure-page";
When Tapestry is determining if a page is secure or not, it starts by checking for the @Secure annotation, then it consults the MetaDataLocator service.
If you want to make your entire application secure:
AppModule.java (partial)

```
public void contributeMetaDataLocator(MappedConfiguration<String,String> configuration)
{
    configuration.add(MetaDataConstants.SECURE_PAGE, "true");
}
```

With no colon, the meta data applies to the entire application (including any component libraries used in the application).

Base URL Support

When Tapestry switches back and forth between secure and unsecure mode, it must create a full URL (rather than a relative URL) that identifies the protocol, server host name and perhaps even a port number.
That can be a stumbling point, especially the server host name. In a cluster, behind a fire wall, the server host name available to Tapestry, via the HttpServletRequest.getServerName() method, is often *not* the server name the client web browser sees ... instead it is the name of the internal server behind the firewall. The firewall server has the correct name from the web browser's point of view.
Because of this, Tapestry includes a hook to allow you to override how these default URLs are created: the BaseURLSource service.
The default implementation is based on just the getServerName() method; it's often not the correct choice even for development.
Fortunately, it is very easy to override this implementation. Here's an example of an override that uses the default port numbers that the Jetty servlet container uses for normal HTTP (port 8080) and for secure HTTPS (port 8443):
AppModule.java (partial)

```
 public static void contributeServiceOverride(MappedConfiguration<Class,Object>
 configuration)
 {
    BaseURLSource source = new BaseURLSource()
    {
       public String getBaseURL(boolean secure)
       {
          String protocol = secure ? "https" : "http";

          int port = secure ? 8443 : 8080;

          return String.format("%s://localhost:%d", protocol, port);
       }
    };

    configuration.add(BaseURLSource.class, source);
 }
```

This override is hardcoded to generate URLs for localhost; as such you might use it for development but certainly not in production.

Content Type and Markup:

Tapestry reads well-formed XML template files and renders its output as XML, with minor caveats:

- The <?xml?> XML declaration is omitted.
- Most elements render with an open and close tag, even if empty.
- Certain elements will be abbreviated to just the open tag, if empty:
  - br
  - hr
  - img
- <![CDATA[]> sections are **not** used

This is all to ensure that the markup stream, while (almost) well formed, is still properly understood by browsers expecting ordinary HTML. In fact, Tapestry may decide to render a purely XML document; it depends on the content type of the response.
When Tapestry renders a page, the output content type and charset is obtained from meta data on the page itself. Meta data is specified using the @Meta annotation.

Content Type

The response content type is obtained via meta-data key tapestry.response-content-type. This value defaults to "text/html", which triggers specialized XML rendering.
A page may declare its content type using the @ContentType class annotation. Content types other than "text/html" will render as well-formed XML documents, including the XML declaration, and more standard behavior for empty elements.

Injection:

Injection is Tapestry's way of making a dependency – such as a resource, asset, component, block or service – available in a page, component, mixin or service class

Injection is a key concept in Tapestry, and it is used in several different but related ways, listed below.

Injection in Tapestry IOC Services

The Tapestry IoC container makes use of injection primarily through constructors and via parameters to service builder methods.

Injection in Component Classes

For components, however, Tapestry takes a completely different tack: injection directly into component fields.
The @Inject annotation is used to identify fields that will contain injected services and other resources.
Tapestry allows for two kinds of injection:

- **Default injection**, where Tapestry determines the object to inject into the field based on its type.
- **Explicit injection**, where the particular service to be injected is specified.

In both cases, the field is transformed into a read only value. As elsewhere in Tapestry, this transformation occurs at runtime (which is very important in terms of being able to test your components). Attempting to update an injected field will result in a runtime exception.
In addition, there are a few special cases that are triggered by specific field types, or additional annotations, in addition, to @Inject, on a field.

Block Injection

For field type Block, the value of the Inject annotation is the id of the <t:block> element within the component's template. Normally, the id of the block is determined from the field name (after stripping out any leading "_" and "$" characters):
@Inject

```
private Block foo;
```

Where that is not appropriate, an @Id annotation can be supplied:

```
@Inject
@Id("bar")
private Block barBlock;
```

The first injection will inject the Block with id "foo" (as always, case is ignored). The second injection will inject the Block with id "bar".

Resource Injection

For a particular set of field types, Tapestry will inject a *resource* related to the component, such as its Locale.
A very common example occurs when a component needs access to its resources. The component can define a field of the appropriate type and use the @Inject annotation without a value:

```
@Inject
private ComponentResources resources;
```

Tapestry uses the type of the field, ComponentResources, to determine what to inject into this field.
The following types are supported for resources injection:

- **java.lang.String** – The complete id of the component, which incorporates the complete class name of the containing page and the nested id of the component within the page.
- **java.util.Locale** – The locale for the component (all components within a page use the same locale).
- **org.slf4j.Logger** – A Logger instance configured for the component, based on the component's class name. SLF4J is a wrapper around Log4J or other logging toolkits.
- **org.apache.tapestry5.ComponentResources** – The resources for the component, often used to generate links related to the component.
- **org.apache.tapestry5.ioc.Messages** – The component message catalog for the component, from which localized messages can be generated.

Asset Injection

When the @Path annotation is also present, then the injected value (relative to the component) will be a localized asset.

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

Symbols in the annotation value are expanded.

Service Injection

Here, a custom EmployeeService service is injected, but any custom or built-in service may be injected in the same way.

```
@Inject
private EmployeeService employeeService;
```