

Tapestry 5

IN ACTION

Igor Drobiazko



MANNING



**MEAP Edition
Manning Early Access Program
Tapestry 5 in Action version 2**

Copyright 2011 Manning Publications

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

Part I: Tapestry for beginners

1. Introducing Tapestry
2. Tapestry templates
3. Page navigation
4. Developing stateful applications
5. Localization
6. Creating forms with Tapestry
7. Converting and validating user input
8. Generating UI for JavaBeans
9. Streaming multimedia content

Part II: Tapestry for advanced users

10. Developing reusable components
11. Mixins
12. Ajax
13. Integrating Hibernate and Spring
14. Testing Tapestry applications
15. Securing your applications
16. Type Coercion

Part III: Tapestry for experts

17. Tapestry IoC and Dependency Injection
18. Applying patterns with Tapestry IoC
19. Creating modular web applications
20. AOP with Tapestry
21. Request processing pipeline in detail
22. Meta-programming with Tapestry

Appendices

- A: Setting up a development environment
- B: Component reference

Introducing Tapestry



This chapter covers

- How Tapestry improves developers' productivity
- Concepts behind Tapestry
- Structure of a Tapestry application
- Naming conventions used in Tapestry to avoid XML configuration of applications
- Tapestry and Model-View-Controller pattern

You are about to be introduced to a web framework which takes an innovative approach for building web applications. Tapestry is an open-source framework for building web applications in Java. It is designed with developers friendliness in mind; it makes the life of Java developers less stressful and more productive. You will realize it from the first minute using Tapestry.

For example, you will see that Tapestry defines a couple of naming conventions which you need to learn. Following these naming conventions you will be extremely productive because there is no need to configure your application using verbose XML files. Furthermore Tapestry boosts your productivity with a unique class-reloading feature. With Tapestry you change the source code and see the results immediately; long deployment cycles are in the past.

Tapestry was originally created by Howard Lewis Ship around 2000 as an internal framework. Two years later, Tapestry 2 was open sourced and hosted at SourceForge¹. In the very same year, the version 3 became an official member of Apache-Jakarta family². In 2006, after a successful vote of the Apache Board,

Tapestry was promoted to an Apache top level project. From this moment on Tapestry gained popularity and the community began to grow. Released in December 2008, Tapestry 5 represents a completely new code base designed to simplify the Tapestry coding model based on lessons learned in the previous releases.

Footnote 1 <http://sourceforge.net/projects/tapestry/>

Footnote 2 <http://jakarta.apache.org/>

The framework behind Tapestry is built upon the standard Java Servlet API, and so it works in any servlet container or application server. To the application server, a Tapestry application is just a servlet filter servicing the incoming requests. Structurally, a Tapestry application is divided into a set of pages, each constructed from reusable components (see figure 1.1). Like pages, components may contain other components, allowing you to build up a complex structure from simple, reusable parts. So, the Tapestry filter is only responsible for linking the incoming requests from the servlet container to the pages and components of the application.

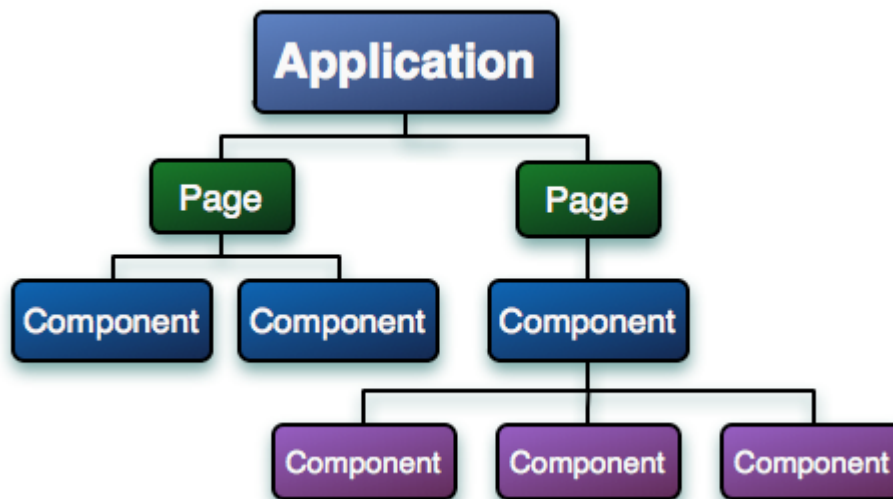


Figure 1.1 Structure of a Tapestry application

The Tapestry framework completely abstracts the Servlet API and provides its own component-oriented programming model on top of it. This way Tapestry lets you code in terms of your objects and the methods and properties of those objects. If you have information to store, store it as fields of your classes, not inside the `HttpServletRequest` or `HttpSession`. If you need some code to execute,

its just a simple annotation or method naming convention to get Tapestry to invoke that method, at the right time, with the right data. The methods don't even have to be public!

In this chapter, we'll provide you with some background on Tapestry, its goals and philosophies, and we'll look at how it works with the Model-View-Controller (MVC) pattern. We'll also go much more into the details on the structure of a Tapestry application and look at Tapestry pages. By the end of this chapter, you'll have a basic overview of Tapestry's capabilities which will be discussed in detail in this book.

1.1 How Tapestry improves developers' productivity

Imagine you were about to design a new web framework for Java. What features would you provide to make the framework successful? Well, the answer is pretty straightforward: improve everything that annoyed you when using competing frameworks. The Tapestry team believes that the most annoying thing when developing web applications with Java is the low developers productivity.

Tapestry is designed with productivity in mind and a lot of features are the result of this design principle. In this section I will show you how Tapestry makes you more productive than any other competing Java web framework by reducing the amount of Java code you need to write, eliminating the deployment time during development and providing error reports when something goes wrong in your code. Let's experience the new level of productivity.

1.1.1 Code less, deliver more

Developing Tapestry applications involves creating HTML templates for pages and components, and combining these templates with small amounts of Java code. Compared to traditional servlet applications or modern competing web frameworks, the code base of Tapestry-based application is much smaller as it primarily contains application-specific logic. How is this accomplished?

First, because the framework behind Tapestry dictates the structure of your application, partitioning it in pages and components. Based on couple of naming conventions, Tapestry knows where Java classes for pages and components are located; there is no need for configuration. You just store your page and component classes in the appropriate place and Tapestry can find them, allowing you to focus on coding, not configuration.

Secondly, the Tapestry framework is non-intrusive, as your code is not expected to conform to the framework. You don't need to extend

framework-specific superclasses or implement predefined interfaces. In Tapestry, the framework adapts to your code. You have control over the names of the methods, the parameters they take, and the value that is returned. Your classes are pure POJOs (Plain Old Java Objects).

Thirdly, your code is not responsible for the plumbing work caused by the environment the application is running in. This is Tapestry's job. For example, you never focus on URLs, extracting query parameters or transforming them to appropriate server-side objects. This is done automatically by Tapestry. You are also rarely confronted with the Servlet API and can program in terms of objects.

1.1.2 Non-stop development

If you have already developed web applications in Java, with any existing framework, you probably will remember the time slots in which you was sitting in front of your desktop and watching the log output of the starting application server. Do you? Now try to estimate how many hours you spent waiting for the server to start up. Unfortunately, too many as you were forced to restart your application server every time you changed any of your Java classes. It is sad, but most Java developers don't really realize this problem, as they simply accept it as a part of the development cycle.

This unnecessary waiting time, when counted over many developers and several months of work, makes your development more expensive. Developers using dynamic languages like PHP or Ruby have a huge advantage over their colleagues using Java. They change their code, switch to the browser, refresh the page and can immediately see the changes. You can think of this as non-stop development.

Enough of that. Tapestry brings non-stop development to the Java world. With Tapestry's unique live-reloading feature you can change the source code, save it and see the results immediately. No redeploy, no restart is required. This way you'll find yourself working with unprecedented high levels of productivity.

To test the live reloading, with the application running, change one of your page classes and reload the page in the browser. Thanks to the combination of background compilation in your favorite IDE and Tapestry watching the classes, templates and resources, the page will immediately change in the browser. You will notice that you can work quite a while without restarting the web server. This gives you an enormously increase in productivity.

1.1.3 Feedback

In most frameworks, open source or proprietary, when something goes wrong in your code, you are on your own. Frequently, you see just a stack trace which tells you very little about the problem. Just recall the last web framework you learned. How many hours did you spend playing Sherlock Holmes?

Tapestry goes to great lengths to protect you from such frustrating situations. An enormous effort has been spent to provide meaningful error reports, which not only tell you that an error occurred, but also give you hints how to solve it.

In figure 1.2 you can see the error report which was caused by an all too common typing mistake in the template. As you can see, it doesn't simply display the exception message, but also additional information to investigate the problem. The message tells you that the template contains an expansion `${hello}` which doesn't match a property in the page class. The only existing properties are `class`, `componentResources` and `hello`. The full template is shown so that you are not forced to switch between the browser and the IDE to get the full picture. The part of the template which caused the error is highlighted. It is clear that the `${hello}` should be changed to `${hello}`.

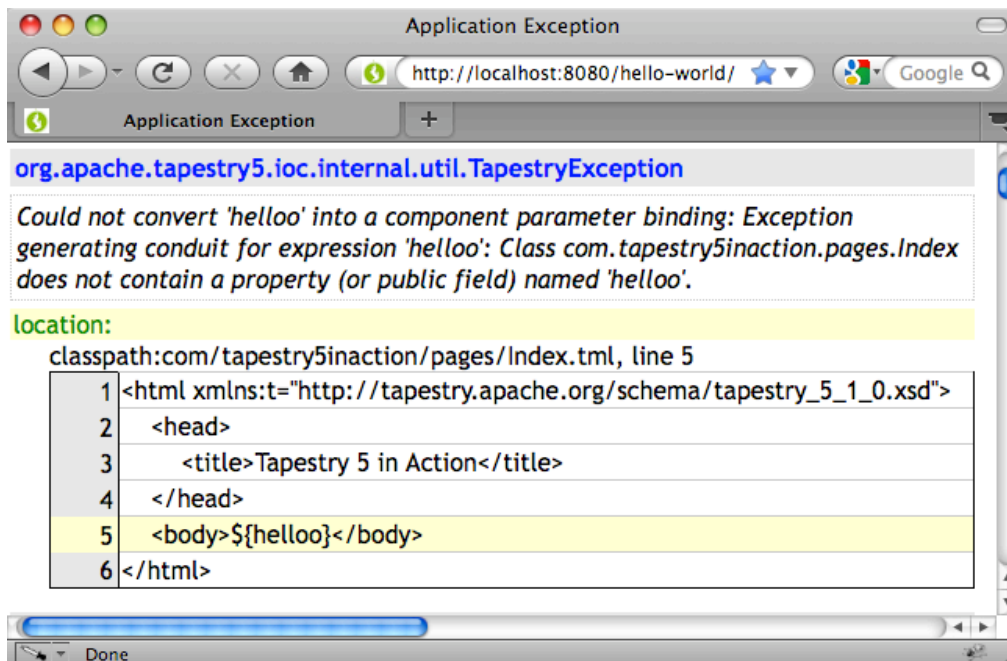


Figure 1.2 Error message caused by a typing mistake in a template. The report tells the developer how to fix the problem by displaying a meaningful message and highlighting the line in the template where error occurred.

Perhaps your template is correct and it's your Java code that has the typo; the fix is just as quick. Just rename the `hello` property in `hello` in your page's

class and Tapestry will pick up this change just as quickly, without the need to restart the application.

The display of such an elaborate error message is only useful during development. You can switch this off when your site is in production. In that case, the error message will not reveal the inner details of your application. How to enable production mode will be explained in chapter XREF ch-di when discussing Tapestry IoC.

1.2 Overview of Tapestry's concepts

Beyond developers productivity there are many more features which should be addressed by a framework like Tapestry. In this section you will learn some fundamental concepts behind Tapestry which have been considered at framework design time. This is a very short summary of what you can expect in the following chapters of this book. Let's dive in.

1.2.1 Object-orientation

In object-oriented (OO) programming, objects have two fundamental characteristics: they are stateful and have behavior. In Java, an object stores its state in fields and exposes its behaviour through methods. Because web-applications are multi-threaded, you have to unlearn object-oriented programming. An application server could be handling thousands of requests from individual users, each in their own thread, and each sharing the exact same objects. You can't store data inside a shared singleton objects, for example servlet, because this data is shared between multiple threads. Any value stored in an instance field for one client will almost immediately be overwritten with a value for another client. Frequently such errors only appear in production as there's usually only a single client during development time.

Tapestry shields you from the multi-threaded aspects of web application development. Tapestry manages the life-cycle of your page and component objects, reserving particular objects to particular threads so you never have to think twice about threading issues. In Tapestry you still can program in terms of object-orientation without caring about the environment your objects are living in.

1.2.2 Developer friendliness

Tapestry is a Java framework, so its ambition is to make life of Java web developers easier. To get started with Tapestry, you only need a Java IDE for writing Java code and a WYSIWYG editor for templates. The competing frameworks are praising the existing graphical tools which simplify development and deployment of web applications. These tools are reputed to enable managers to create web applications without any programming skills, just by clicking buttons and using drag-and-drop functionality. Tapestry's approach is different: the framework was designed with developer friendliness in mind, so that rich graphical tools are needless.

Everybody is speaking about the Separation of Concerns principle, but only few technologies provide you the capability to enforce it. With Tapestry you can work on a specific aspect of your application without having significant impact on other parts of the application. For example, a HTML designer can create templates for pages and components because they are so close to ordinary HTML, without all the cruft and confusion seen in JavaServer Pages. Java developers can focus on application logic, relying on their HTML gurus for the design.

1.2.3 Avoiding XML configuration

Many Java frameworks make extensively use of XML to describe the configuration of the applications written with these frameworks. Probably the most prominent examples are Struts, JavaServer Faces and Spring. The main problem with XML configuration files is that those often reference class names, or even method names, that may be refactored (renamed or moved) and the IDE may not be able to keep the files synchronized. Furthermore, because XML is quite verbose, sooner or later the configuration files will be hard to maintain.

For most Java programmers it is far more natural to write Java code than to declare the behaviour of the application in XML configuration files. XML is just a suboptimal way to describe Java operations like instantiating classes and invoking methods on those classes. So, one of the central concepts in Tapestry 5 is to eliminate XML and build an equivalent system around simple objects and methods. This system is called Tapestry IoC, a built-in Inversion of Control container. Based on a handful of naming conventions and annotations Tapestry IoC allows you to configure of your application programmatically, instead of declaring a big chunk of XML.

1.2.4 Inversion of control

As you already learned, page and component classes in Tapestry are pure POJOs (Plain Old Java Objects). Contrary to most other web frameworks, you are not required to extend a framework specific superclass or implement an interface to be able to access the framework's functionality. Tapestry uses a different approach allowing you to inject the needed functionality into pages and components. The inner construction of the Tapestry framework is based on Inversion of Control (IoC), a design approach that allows a system to be fabricated from many small pieces. Tapestry IoC (Inversion of Control) is a sub-framework of Tapestry that acts as a container for objects, known as services. Don't worry if you are not familiar with Inversion of Control, in chapter XREF ch-di you will get a complete explanation with many examples. For now it is sufficient to consider Tapestry IoC as a part of the framework which magically injects services, pages or components into your classes. This injection is guided by annotations, as shown in the following example.

Listing 1.1 Injecting services, pages and components

```
public class InjectDemo {

    @InjectPage
    private MyPage page;

    @InjectComponent
    private MyComponent component;

    @Inject
    private HttpServletRequest request;

    ...

    Object onSuccess() {
        return page;
    }
}
```

- ❶ Injecting page instance
- ❷ Injecting component instance
- ❸ Injecting service

The `@InjectPage` annotation ❶ is used to inject a page instance, while `@InjectComponent` ❷ injects a component into a page or component. Using the `@Inject` annotation ❸ gives you access to any service built into Tapestry or implemented by you. In this example, an instance of `HttpServletRequest` is

injected into the page. See how elegant this solution is. There is no pollution of your class, you are free to have any super class and don't need to override any inherited methods.

1.2.5 Testability

Typically the code of Tapestry applications is broken into small pieces, each for a well-defined functionality. The built-in IoC container is responsible to put all those tiny bits back together into a running application. However, breaking an application into smaller pieces makes it much easier to test. As Tapestry doesn't force you to extend your pages and components from any framework classes, it is extremely simple to write tests. You can just instantiate your page or component class, provide needed services as mocks, invoke methods on created instances and assert the results. Tapestry shields you from the usage of the Servlet API, you don't need to start any server to write a simple unit test.

The following screen shot demonstrates a result of the Tapestry's testability. As you can see, the test coverage of the Tapestry's core library is 92 percent. This relatively high coverage is only possible due to Tapestry's test friendly design. If you care about testability, you'll love Tapestry's approach to develop web applications. Tapestry doesn't limit you when it comes to testability but even provides you great testing facilities which allow you to achieve a respectable test coverage of your application, similar to Tapestry itself.

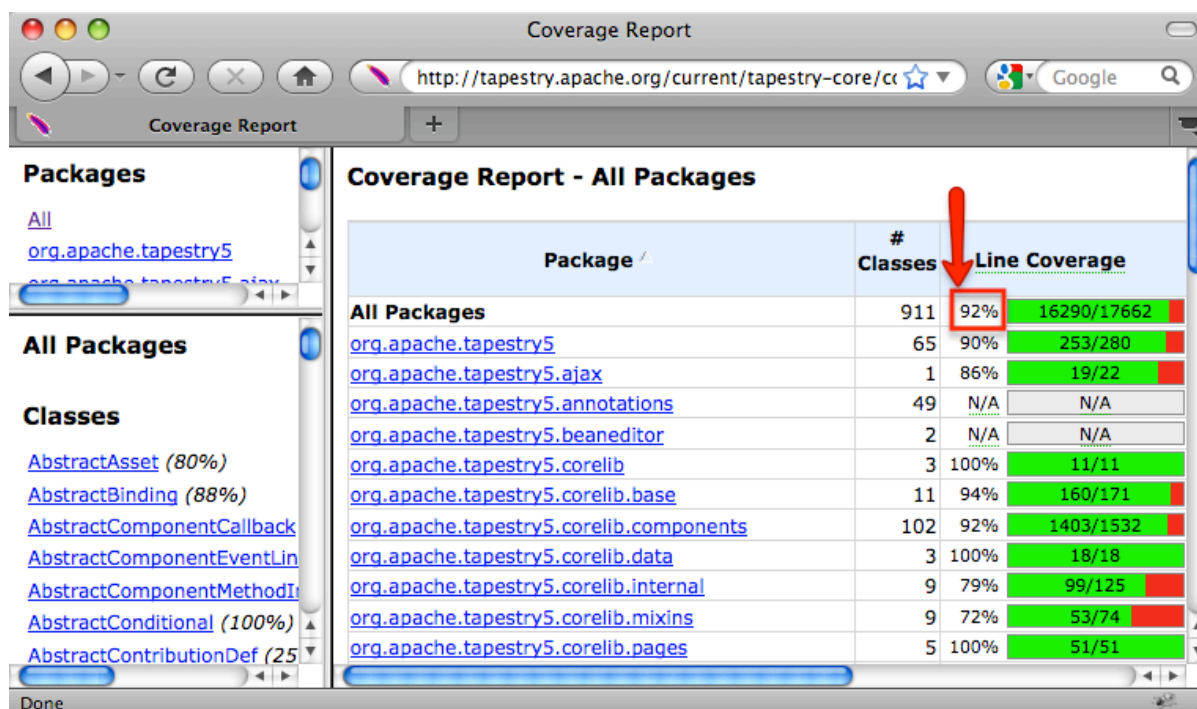


Figure 1.3 Tapestry's test coverage report used by Tapestry developers to measure the

percentage of Tapestry code accessed by tests. The report tells that 92 % of Tapestry' code has been covered by tests. Such a high test coverage is an indicator for Tapestry's quality.

1.2.6 Scalability

Tapestry was designed with scalability in mind. Applications written with Tapestry may contain a large number of pages, each structured into embedded components. The structure of any particular page is static; once defined at development time this structure can not be changed at runtime. Creating a page instance is an very costly process as it is the root of a large tree of other objects including embedded components, many kinds of structural objects, template objects, and others. Recreating page instances for each request is not performant, thus a way to reuse page instances is needed. Competing frameworks like JavaServer Faces or Wicket make an extensive use of `HttpSession`, storing the entire component hierarchy into it. This way the component hierarchy is more dynamic at cost of scalability. In a clustered environment any data stored in the session needs to be replicated around the cluster nodes. So, persisting the component hierarchy into the session causes a serialization and distribution overhead, which can and should be avoided. Tapestry solves this challenge in a different way.

Before Tapestry 5.2, the page objects were pooled. Because of the static structure of a page, all the instances of that page inside the pool are equivalent. This also works in a clustered environment; all page instances on all servers within a cluster are uniform. For an incoming request Tapestry can use any of the available page instances, or create a new one as needed. Once created, a page instance will be stored in the pool and reused in later requests. When a request comes in, a page instance is taken from the pool and attached to the request exclusively. After the request has been handled, the reserved page instance returns to the pool.

When a page instance is attached to the current request, it is enriched by the values specific to that request. These values are typically stored in `HttpSession`. The big difference with competing frameworks, is that only these values are stored in the session. There is no need to store the entire page instance along with the huge component structure in the session. This is a more scalable approach.

As you can imagine, maintaining a pool of page instances for very big applications can exhaust the available resources on the server. In fact, several Tapestry users reported heap space problems in applications with big pages, consisting from thousands of components. For this reason the page pool has been

deprecated in Tapestry 5.2. Instead, all Tapestry pages are now singletons. That single page instance can be used across multiple threads because Tapestry maintains a per-thread Map for the current request. As part of the class transformation process, all mutable fields of a page are converted into an access against this per-thread Map. The end result is that a single page instance can be used across multiple threads without any conflicts.

1.2.7 Plug-and-play nature

The framework behind Tapestry is designed to build modular web applications. The built-in IoC container allows you to break a monolithic WAR file into several modules, each responsible for a certain functionality. These modules can be used as lego bricks to assemble an application from independent JAR files. Tapestry's modularity facilities can be used to extend an existing application with new functionality without changing the code of the application. For example you might create a module which exposes the functionality of your web application as web services. Tapestry supplies the facility which allows you to drop a new module on the classpath, restart your server and immediately use the new functionality.

Another frequent use case is delivering the same application to several customers. In such a case you often have a problem that a special requirement of a customer conflicts with a special requirement of another customer. If you include these special wishes into the application, it will become bloated sooner or later. What you need is a feature that allows you to extend your application without changing it. This is where your decision to write web application with Tapestry pays off. Such a requirement is amazingly easy to implement using Tapestry, but might be a huge problem with other frameworks.

Now that we have learned the basic concepts behind Tapestry, let's explore a typical structure of a Tapestry application. Before continuing you might need to set up your development environment, if not yet done. Please refer to appendix XREF `appendix_setting_up_development_environment` for instructions on how to get your first Tapestry application up and running.

1.3 The structure of a Tapestry application

Tapestry applications are standard Java web application packaged into a WAR (web application archive) file. A web application archive is actually a normal JAR file that has a specific directory structure, as shown in figure 1.4 .

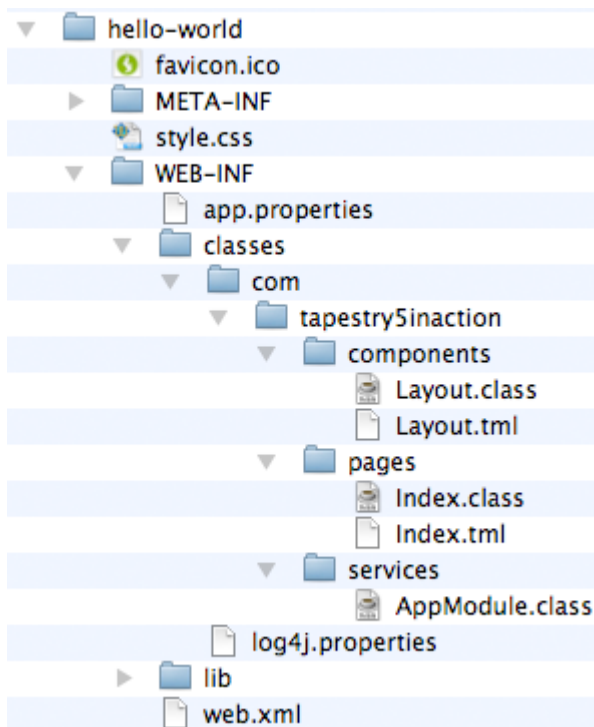


Figure 1.4 Structure of a Tapestry application

The top-level directory of a WAR file is used to store static files like images, Cascading Style Sheets, JavaScript files, etc. The subdirectory *WEB-INF* is special; it is expected to contain the following files and directories:

- *web.xml* - Web application deployment descriptor.
- *classes* - Directory containing compiled Java classes of the web application.
- *lib* - Directory containing further JAR archives the web application depends on.

So far, we discussed a standard structure of a web application archive; now let's explore Tapestry-specific details. First we will have a look at the web deployment descriptor named *web.xml* file (listing 1.2) inside the *WEB-INF* folder of the application. This XML file is a standard descriptor defined in the Servlet API to provide the configuration and deployment information for a web application. Listing 1.2 shows a typical *web.xml* descriptor for a Tapestry application.

Listing 1.2 *web.xml*: web deployment descriptor

```
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
```

```

"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Tapestry 5 Application</display-name>

  <context-param>
    <param-name>tapestry.app-package</param-name>
    <param-value>com.tapestry5inaction</param-value>
  </context-param>

  <filter>
    <filter-name>app</filter-name>
    <filter-class>org.apache.tapestry5.TapestryFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>app</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
</web-app>

```

① Application's root package

② Filter name

③ Filter mapping

Two aspects are important: the context parameter `tapestry.app-package` ① and the name of the Tapestry filter ②. The Tapestry filter is an implementation of the `javax.servlet.Filter` ³ interface, which takes the role of central controller in many web frameworks. This filter is an entry point into a Tapestry application; it identifies the requests that are relevant to Tapestry ③, and lets the servlet container handle the rest. The filter is also responsible to initialize Tapestry on application startup. As you will learn later, the name of the Tapestry filter is used to identify the class name of the IoC module for the application, the so called *application module*. This is elaborated on in chapter XREF ch-di. For now it is sufficient to know that the class name `AppModule` in package `com.tapestry5inaction.services` is constructed from the filter name `app` and the suffix `Module`. This is the first naming convention used by Tapestry which you need to remember.

Footnote 3 <http://www.oracle.com/technetwork/java/filters-137243.html>

Unlike nearly all Java web frameworks, Tapestry doesn't require any additional XML configuration files except *web.xml*. The only configuration you need to provide is the `tapestry.app-package` context parameter. The value of this parameter is interpreted as the name of the root package for the application, which may contain predefined sub-packages for different artifacts of an application:

- *base* - Sub-package for the base classes. This can contain superclasses for pages and components.

- *components* - Sub-package for the component classes.
- *entities* - Sub-package for Hibernate entities (see chapter XREF ch-hibernate-spring).
- *mixins* - Sub-package for mixins (see chapter XREF ch-mixins).
- *pages* - Sub-package for page classes.
- *services* - Sub-package for the *application module* class. The services are also located in this package.

According to the naming convention, pages will be in the `pages` sub-package and components in the `components` sub-package. In listing 1.2 the root package for the application is `com.tapestry5inaction`. So, Tapestry will look for page classes inside the package `com.tapestry5inaction.pages`. Accordingly the package for components is `com.tapestry5inaction.components`.

The sub-packages named above are called *controlled packages*. Tapestry observes classes, templates and resource files in these sub-packages for changes and reloads them as soon as any change made by you is detected. This way Tapestry provides the unique class reloading feature mentioned in section 1.1.2. This feature allows you to change the source code and see the results immediately without to restart your application. Note that there are some limitations on live realoding in sub-packages `entities` and `services`, which will be discussed in chapters XREF ch-hibernate-spring and XREF ch-di.

NOTE**Take care what you store and where**

You should be careful about what you store into controlled packages as each of them is dedicated to a certain functionality. For example you should never store other classes than page classes into `pages` sub-package. Tapestry will identify them as page classes and perform the page-specific class enhancements. For example, placing domain objects into this package might lead to unexpected results.

Of course, you are free to create other packages inside your web application to store other artifacts like utility classes. Tapestry doesn't limit you at all.

Now that you are familiar with the structure of Tapestry applications, let's get started by creating your first page.

1.4 Creating a Tapestry page

A Tapestry page is a duo, consisting of a Java class and a template, which may be accessed through a URL. The Java class is the required part of the page, whereby a template is optional. Contrary to most other web frameworks the page classes are POJOs (Plain Old Java Objects). The framework does not force you to extend a superclass or implement a predefined interface to access the framework's functionality. In Tapestry the needed functionality is implemented in a set of services which can simply be injected using the built-in IoC container (see chapter XREF ch-di).

The second (and optional) part of a Tapestry page is a template. Templates are well-formed (X)HTML files which can be recognized by the "*.tml" extension. TML stands for *Tapestry Markup Language* and is comparable to JSP (JavaServer Pages). An advantage of a Tapestry template compared with JSP is that the template is not compiled to a servlet by the web container. A Tapestry template does not allow mixing of markup and Java code (as is possible in JSP), forcing a clean split between view and controller. As you will see shortly, templates can be previewed without starting a server.

The link between a page class and template is based on the file name. For each page class, a template file with the same name is searched in the same package. For example for the page class `Page.class` the template is named `Page.tml`. So both files are expected to be located on the classpath in the package `com.tapestry5inaction.pages`. Tapestry also allows you to store the templates for pages (not components) into the root folder of a WAR file because it is a common place for the static web content and you might be used to store you JSPs in that folder. It is highly encouraged to place the templates on the classpath as this is more consistent. Storing templates inside the application context might be deprecated in the future releases.

Let me show you how to write simple pages, with templates and without. First let's look at the code for the start page of the "Hello World" application in listing 1.3 . As you can see, this is a pure POJO inside the `pages` sub-package. Do you remember that package name? This is a sub-package for page classes.

Listing 1.3 `Index.java`: Java class for the start page

```
package com.tapestry5inaction.pages;  
  
public class Index {
```

```

public String getHello() {
    return "Hello, World!";
}
}

```

The goal in Tapestry is for templates to look as much as possible like ordinary HTML files. In fact, the template for the `Index` page (listing 1.4) is well-formed (X)HTML file. Let's explore it.

Listing 1.4 Index.html: template for the start page

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_1"
  <head>
    <title>Tapestry 5 in Action</title>
  </head>
  <body>
    ${hello}
  </body>
</html>

```

① **Tapestry namespace**

② **Expansion**

The `xmlns:t` attribute on the first line connects the `t` prefix to the Tapestry schema ①. This prefix is used to access Tapestry' built-in components and template elements. Tapestry's internal template parser recognizes that URL and can resolve XML tags to Tapestry components. In this example the prefix is not used yet. The `${hello}` ② expression is called an *expansion*. Expansions can be compared with EL (Expression Language) in JavaServer Pages. They are used to provide the static templates with dynamic behaviour. The expression between the braces is evaluated when the template is used to create the markup for the response. In this case, the expansion causes Tapestry to invoke the `getHello()` method on the page instance, as shown in figure 1.5. So this expansion causes the string "Hello, World!" to be printed between `<body>` and `</body>`. You could consider the expansion to be a placeholder which is replaced by the evaluation of the expression at run-time. More details on expansions are coming in chapter XREF ch-quickstart .

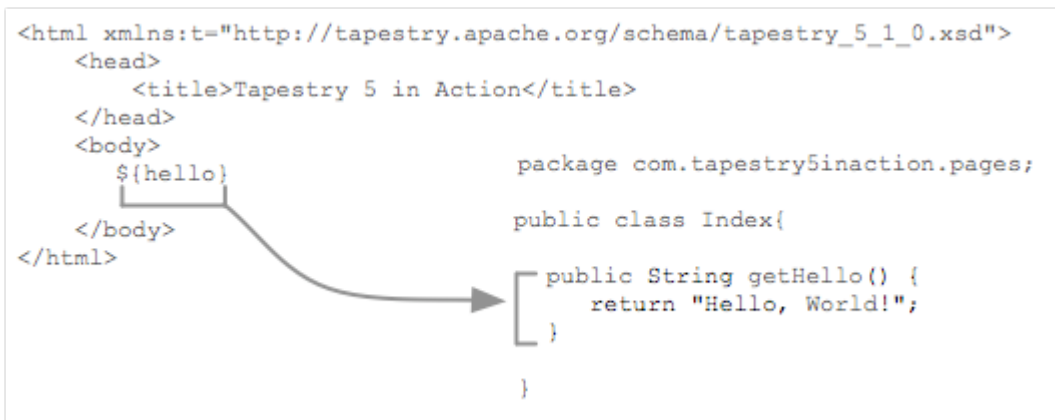


Figure 1.5 The expansion `${hello}` is evaluated at runtime. It tells Tapestry to invoke the `getHello()` method in the page's Java class.

Every page inside a Tapestry application can be accessed through a URL, as shown in figure 1.6. The page from listing 1.3 will be accessible using the URL <http://localhost:8080/hello-world/index>, if the context path is "hello-world". Tapestry interprets the part of the URL after the context path as page name. The extracted page name will be used to locate the page class inside `pages` sub-package which will be used along with the corresponding template to render the response. The URLs in Tapestry are case-insensitive, so that the URL <http://localhost:8080/hello-world/InDeX> will access the same page. The page named `Index` has a special meaning in Tapestry; it is the start page for the application. When the request URL does not contain a page name, the `Index` page is displayed. This matches the default behavior for most web servers.

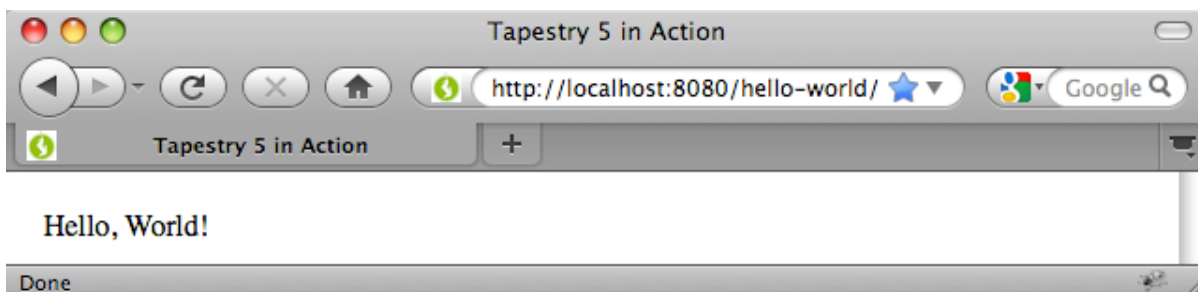


Figure 1.6 Your first Tapestry page

You can also create sub-packages in the `pages` package to organize your application when there are many pages. In that case, the sub-package will also appear in the URL. The page `com.tapestry5inaction.pages.view.Details` would be accessible using the URL <http://localhost:8080/hello-world/view/details>.

As already mentioned, a template is optional for a Tapestry page. You can also

produce the markup in Java code using the Tapestry API. As a little teaser, listing 1.5 shows a page without template. More details follow in chapter XREF [ch-reusable-components](#) .

Listing 1.5 PageWithoutTemplate.java: this page doesn't need a template as the markup is produced in Java code

```
package com.tapestry5inaction.pages;

import org.apache.tapestry5.MarkupWriter;

public class PageWithoutTemplate {

    void beginRender(MarkupWriter writer) {
        writer.element("html");
        writer.element("body");
        writer.writeRaw("Hello, World!");
        writer.end();
        writer.end();
    }
}
```

1 Begins html tag

2 Writes text

3 Ends html tag

Now that we learned how to write a simple Tapestry page, let's explore Tapestry components.

1.5 Using and creating Tapestry components

Components are reusable blocks of code for a Tapestry application, allowing functionality to be reused in several pages. As already mentioned Tapestry pages are often structured into components which encapsulate special reusable functionality. In this section I'll show you how to use built-in Tapestry components and how to create your own components. Before you learn how to write your first component, let's take a look at how the components are used in a page.

1.5.1 Using the If component

Tapestry provides a set of components which can be used out of the box. Let's start with the a very simple component which enables you to make two-way decisions in a page, the `If` component. This component is a good start to learn how Tapestry components are used in a page because two-way decisions are so common, no matter what background knowledge in programming languages or web frameworks you have. Imagine you want to display a welcome message to a user only if he or she is logged in.

Listing 1.6 Using the If component

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <head>
    <title>Tapestry 5 in Action</title>
  </head>
  <body>
    <t:if test="user">
      Welcome, ${user.firstName}
    </t:if>
  </body>
</html>

```

In the template above you can see the `<t:if>` tag which is a non-standard HTML element. The `xmlns:t` defined in the root element of the template connects the `t:` prefix with Tapestry's schema. All tags associated with that namespace are considered as Tapestry components. So, when the template above is parsed, Tapestry identifies the `<t:if>` tag as the `If` component which is built into Tapestry's core library and creates a functioning component instance. First Tapestry resolves the component class from the tag name. In case of `<t:if>` the class name is `If`. But where is this class located? You learned already that Tapestry applications have a root package with predefined sub-packages. However, any JAR file inside your application may contain its own root package. Such JAR files are called component libraries (see chapter XREF ch-reusable-components). The root package for the Tapestry's core library is `org.apache.tapestry5.corelib`. So, according to the conventions for sub-packages the class of the `<t:if>` component is located inside the `org.apache.tapestry5.corelib.components` package. Once the component class is found, Tapestry instantiates an instance of it and performs some other operations to assemble a functioning component instance.

A component may have any number of parameters that can be used to configure a component instance. For example the `If` component has a required `test` parameter which specifies a conditional expression to evaluate. In the example above the expression is a reference to the `user` property in the page classes. If the value of the property is non-null, the expression is evaluated to `true`, otherwise to `false`. In case of `true` value, the body of the `<t:if>` tag is printed.

An alternative way to define a component is to create an instance variable of component's type in the page class and to place the `@Component` annotation on that property. The `parameters` attribute of the `@Component` annotation is used to pass parameter values to the component.

Listing 1.7 IfDemo.java: Defining component with `@Component` annotation

```

package com.tapestry5inaction.pages;

import org.apache.tapestry5.annotations.Component;
import org.apache.tapestry5.corelib.components.If;

import com.tapestry5inaction.entities.User;

public class IfDemo {

    @Component(parameters = "test=user")
    private If iff;

    public User getUser() {
        return ...;
    }
}

```

In order to use this instance of the `If` component in the page's template we need to link a tag with that component. This is done using the `t:id` attribute. This attribute tells Tapestry to treat a tag not as a usual HTML tag but as a component which is defined in the corresponding Java class. The value of `t:id` is interpreted as component's id which by default is derived from the name of the instance variable used to define the component. In the example above the id of the `If` component is `iff`. Now let's see how to reference the component from the page's template.

Listing 1.8 IfDemo.tml: Referencing components defined in the Java class

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <head>
    <title>Tapestry 5 in Action</title>
  </head>
  <body>
    <div t:id="iff">
      Welcome, ${user.firstName}
    </div>
  </body>
</html>

```

SIDEBAR **t:id vs id**

It makes a big difference whether you use `id` or `t:id` in a template. When using `id`, you are setting the client id for the rendered element within the client-side DOM. This id can be used for client-side logic, for example to access the HTML element from a JavaScript function.

The `t:id` attribute is the unique Tapestry component id. This is the id you might use to inject the component into your page class using the `@InjectComponent` annotation.

A client id and a Tapestry component id may have different values.

Besides the `If` component, Tapestry provides a set of built-in components (see appendix XREF appendix_component_reference) which will help you to assemble your pages from reusable pieces. The built-in components cover the most frequent use cases, so that you will be able to come far using them. At the latest, when you don't find a built-in component which covers your use case, it is time to build your own component. Building custom components is one of Tapestry's strengths, so let's create your first own component.

1.5.2 How to create your own component

In this section you will get a preview how to create your own components. More details on components follow in chapter XREF ch-reusable-components.

Components are very similar to pages⁴ and consist of a component class and an optional template. The difference between a page and a component is minimal. Most importantly, they have to be placed in separate sub-package of the application root package. Pages are stored in sub-package `pages` and components in sub-package `components`. The component template, if available, should be stored in the same package as the component class. You can define further sub-packages to organize your components.

Footnote 4 Strictly speaking it is the reverse, a page is a special kind of component.

To see how small the difference is, you can convert the `Index` page to a component by copying the class from package `com.tapestry5inaction.pages` to `com.tapestry5inaction.components` and renaming it to `HelloWorldComponent` (listing 1.9).

Listing 1.9 HelloWorldComponent.java

```

package com.tapestry5inaction.components;

public class HelloWorldComponent {

    public String getHello() {
        return "Hello World!";
    }
}

```

You also have to create a template in sub-package `components` named `HelloWorldComponent.tml` (listing 1.10).

Listing 1.10 HelloWorldComponent.tml

```
<span>${hello}</span>
```

Note that the template requires a root element because Tapestry's templates are well formed (X)HTML files; in this example we use the `` tag. The expansion `${hello}` is used to access the `getHello()` method from the component class. If you want to use further components in the template, then you would have to declare the Tapestry namespace as shown in previous examples in this chapter.

The `Index` class can now be modified, removing the `getHello()` method as this has been moved to the `HelloWorld` component.

Listing 1.11 Index.java: Page class when using the HelloWorld component

```
public class Index {}
```

In the template for the `Index` page, we replace the expansion `${hello}` with `<t:helloworldcomponent/>` (see listing 1.12).

Listing 1.12 Index.tml: age class when using the HelloWorld component

```

<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <head>
    <title>Tapestry 5 in Action</title>
  </head>
  <body>
    <t:helloworldcomponent/>
  </body>
</html>

```

Again, the prefix `t:` tells Tapestry that the value after the colon should be interpreted as a component name, namely `HelloWorldComponent`. This name

will be used to find a component class inside the components sub-package. Note that we use the same prefix for our own and built-in into Tapestry components. Later you will learn that components may also be bundled into component libraries, so the lookup after a component is not limited to the components sub-package. However, the components from libraries have other prefixes than `t:`. As almost everything in Tapestry, a component name in a template is case insensitive. You can just as well use `<t:HeLLoWoRldComPonEnt/>` in a page template or name your component class `HELLOWORLDCOMPONENT.java`. Tapestry will still know which component to use. No extra description is needed for the component. You just create the component class and if needed the template and you can immediately start using the component. Now if you start the application, you will see that the page looks as shown in figure 1.6.

This simple example shows how easy it is to write and use components in Tapestry. More details on writing components can be found in chapter XREF [ch-reusable-components](#).

1.5.3 Creating a layout component

Many Java web frameworks use specific other frameworks like SiteMesh or Tiles to skin the application. In Tapestry this can also easily be done using a simple component. Let's build a `Layout` component which contains the common content for all pages like title, header, footer and navigation, wrapping itself around the page specific content. In the `com.tapestry5inaction.components` package we create the class `Layout`. This is an empty class which only contains the `@Import` annotation. This annotation automatically includes a Cascading Style Sheet (CSS) file `style.css` when rendering a page wrapped by `Layout` component. The `context:` prefix indicates that the file can be found in the web context directory. Alternatively you can also use the `classpath:` prefix to indicate that the resource should be found on the classpath.

Listing 1.13 Layout.java

```
package com.tapestry5inaction.components;

import org.apache.tapestry5.annotations.Import;

@Import(stylesheets = "context:style.css")
public class Layout { }
```

In the same package, where the `Layout.java` file is located, you have to create

the template for the component (listing 1.14). Let's copy the content of the `Index` page into the `Layout.html` file. Now the specific content for the `Index` page should be replaced by `<t:body/>`. This element serves as a placeholder for the decorated page and will be replaced by the page specific content at runtime.

Listing 1.14 `Layout.html`

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <head>
    <title>Tapestry 5 in Action</title>
  </head>
  <body>
    <t:body/>
  </body>
</html>
```

We can now update the template for our `Index` page by removing the HTML skeleton around the page content and using the newly created `Layout` component. The decorated page template is shown in the following example.

```
<t:layout xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  ${hello}
</t:layout>
```

As already mentioned, you should keep your templates as close as possible to ordinary HTML. The example above doesn't really look like a web page; let's improve it by using the facility called *invisible instrumentation*. The following template is equivalent to the example above.

```
<html t:type="layout"
  xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  ${hello}
</html>
```

As you can see we replaced the `<t:layout>` component by an ordinary `<html>` element which makes the template look like an ordinary HTML file. What happened to the `Layout` component? The standard `<html>` element is attached to the `Layout` component using the `t:type` attribute. When the page is rendered, the `<html>` tag will be replaced by the content from the `Layout.html` template. More details on invisible instrumentation follow in chapter [XREF ch-quickstart](#).

Additionally, you can make use of `<t:content>` element to support the WYSIWYG preview of a page template. This element marks a portion of the

template as the actual template content; any markup outside the `<t:content>` element is ignored when the page is rendered. This is useful if you want to provide an alternative layout for your page that is only used for a WYSIWYG preview. The listing 1.15 demonstrates how to support a preview of a template by providing an alternative HTML skeleton used for a WYSIWYG preview. In production the markup outside the `<t:content>` element will be ignored.

Listing 1.15 Index.tml: Supporting WYSIWYG preview

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_1_0.xsd">
  <head>
    <title>Tapestry 5 in Action</title>
  </head>
  <t:content>
    <body t:type="layout">${hello}</body>
  </t:content>
</html>
```

Now that you have a basic knowledge on Tapestry, let's explore the Model-View-Controller (MVC) pattern. The MVC is the most widely used architectural approach for building web applications in Java. Almost every existing Java web framework implements this pattern, also Tapestry does so. Let's explore the MVC architecture and Tapestry's approach to implement it.

1.6 Tapestry and the Model-View-Controller pattern

Model-View-Controller (MVC) is a design pattern which appeared in the eighties in Smalltalk for the implementation of graphical user interfaces. The MVC pattern splits each user interaction in three aspects, as seen in figure 1.7 :

1. Model - Component which contains the data which needs to be displayed to the user. The model is the contract between view and controller and is usually implemented as JavaBeans.
2. View - Component which displays the data from the model, as prepared by the controller.
3. Controller - Component which reads the data from the model and prepares it for display by the view. The controller also used to interpret the user input which is necessary for the execution of the application logic. As a result of input, the model may be updated.

In a desktop application in which the user interface is built using Swing or

Eclipse Rich Client Platform, there are often different views which offer alternative presentation of the data in the model. When the model changes, all views need to be updated. In traditional MVC this is typically done using the observer pattern. In this pattern, the views register themselves as listeners to changes in the model. When the client sends a request, the input data is interpreted by the controller. This triggers an update of the model which is then passed on to the view.

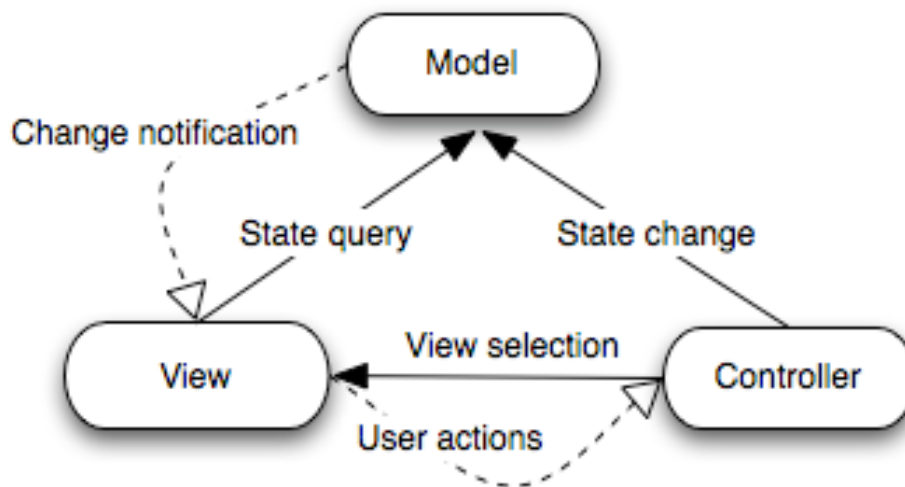


Figure 1.7 Model-View-Controller

Advantages of MVC include:

1. Separation of Concerns.
2. Code reusability.
3. Centralisation of controller.
4. Extensibility of the application.

As HTTP is a pull protocol, the observer pattern does not work on the web. When a server has not received a request, it cannot send information to the client (push). To overcome this limitation, a modified version of MVC is applied on the web, in which the view pulls the data from the model. The controller is still responsible for the link between model and view, handling the input data which is entered and manipulating the application state. In effect, the view is decoupled from the model. The view only knows where the data can be fetched and how changes can be sent back. This results in the 3-tier architecture as seen in figure 1.8

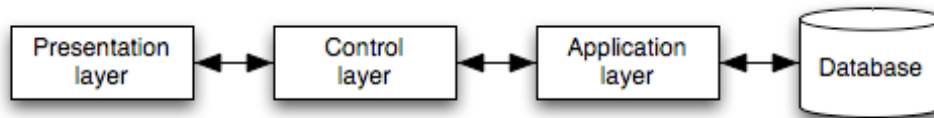


Figure 1.8 3-tier architecture

As in many other MVC frameworks, the application specific domain objects fulfill the role of the model in a Tapestry application. They are designed and implemented by the developer of the specific application.

The role of the central controller is handled in Tapestry by a servlet filter ⁵, which is configured in the web.xml web deployment descriptor. This filter investigates each request and amongst others, chooses the controller for handling the request.

Footnote 5 <http://java.sun.com/products/servlet/Filters.html>

In Tapestry applications the controller logic is implemented in so-called handler methods in the component and page classes. This makes component classes controllers. A handler method responds to an event, which can be triggered by user interaction. For example, when a link is clicked or a form submitted, Tapestry triggers an event. You can handle this event by providing a handler method in a component. Based on a naming convention (see chapter XREF ch-quickstart) or annotations Tapestry will find the proper method to handle the event.

Contrary to many other MVC frameworks (like Struts) Tapestry does not force you to map request URLs to controllers in predefined configuration files. You never have to worry about the URLs and the request parameters. In Tapestry each incoming request contains the page name, possibly the id of a component in the page and (when needed) the so-called *context*. You do not have to decode the URL, Tapestry does that for you. You just have to implement a handler method, which is invoked by Tapestry automatically.

In Tapestry, a page is only marginally different from a component, so pages also have the role of controller in the MVC pattern.

To display data, coming from the model, Tapestry uses a template. The template is the view in the MVC pattern. A template connects to the model using *expansions* (see section 1.4). Expansions are similar to EL (Expression Language) in JSPs. They allow you to bind to the properties of the model into component parameters. Using *dot-notation*, pages and components can navigate the properties of the model, reading or writing the value.

1.7 Introducing the demo applications

As the title of this book implies, the aim of the book is to teach you Tapestry in a very practical way. Because I want the reader to make his/her hands dirty right from the beginning, I created three demo applications, each serving its own purpose. The applications can be downloaded from the book's website <http://www.manning.com/drobiazko> and directly deployed into the servlet container of your choice. The source code for the applications is available at <https://github.com/drobiazko/tapestry5inaction>.

The "Hello World" application is a blank application containing a single page. You can use this application to make your first steps with Tapestry. For example you can browse the source code to become familiar with the typical structure of a Tapestry applications or to start a new Tapestry project.

The "Showcase" application is the official guide for this book. It is an instant, ready to explore application containing the examples from the listings in this book. You can browse the pages of this application and learn Tapestry without to start your IDE. Any example page in this application, does not only display the end result, but also the source code of both page's Java class and template.

Finally, the last application is a simplified clone of Wordpress⁶ created with Tapestry. The application is a blogging software and is called Tlog, which is a blend of Tapestry and blog. It is not recommended, to dive into the source code of this application immediately, as we'll develop some parts of Tlog together through the chapters of this book. The most examples in the "Showcase" application are simplified pages of Tlog, used to prepare you for the advanced application, such as Tlog. Once you have mastered the basics of Tapestry, you can play around with Tlog and see the beauty and power of Tapestry. Let's start with our blog application by creating the domain model.

Footnote 6 <http://wordpress.org/>

The following class diagram shows the basic entities for our blog application. The `Blog` class represents a blog with a name and a description. The `Article` class represents an article to be posted on the blog. An article has a title, a publish date and a content. An article can be commented on and may be tagged. A comment is represented by the `Comment` class and a tag by the `Tag` class. Note that each entity has an `id` field, which represents the primary key of the entity in

the database. According to Tapestry's naming conventions the entities are located in the entities sub-package. In this example the full package name is `com.tapestry5inaction.entities`.

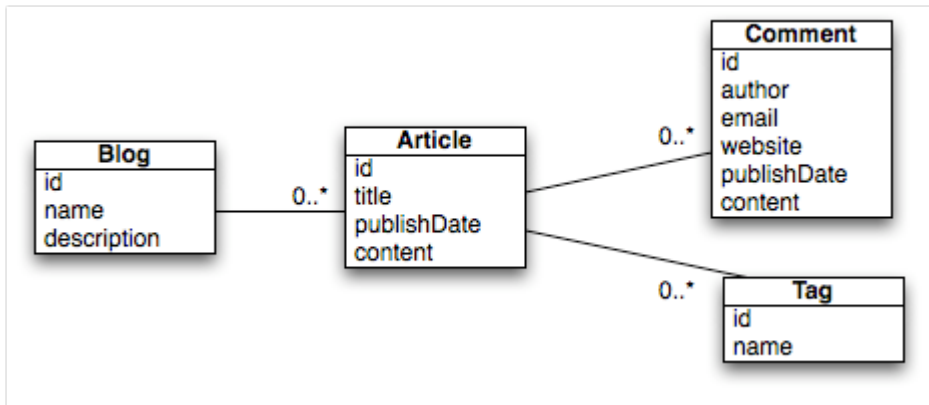


Figure 1.9 Domain model for the Blog application

Now let's define a service for accessing our entities in the database. This service is a typical DAO (Data Access Object) with methods to find, save and delete entities. DAO is a standard JEE pattern for abstraction of accessing data from a database. In listing 1.16 you can see the interface which specifies the required methods. In a production environment, we would provide an implementation using JDBC (Java Database Connectivity) or an ORM-framework (Object Relational Mapping) like Hibernate or JPA (Java Persistence API). For now the implementation details are not interesting.

Listing 1.16 BlogService.java: Basic DAO service to find, persist and delete entities

```

public interface BlogService {

    List<Article> findRecentArticles();

    Article findArticleById(Long id);

    void delete(Article);
}
  
```

1.8 Summary

In this chapter you learned how Tapestry can boost your productivity by reducing the turnaround time spent waiting for the application to be deployed after you changed a single line of code. The next productivity factor are the smart error reports which help you to debug your application when something went wrong. You also have seen that Tapestry reduces the amount of Java code you need to write. This fact makes you more productive again.

We then took a quick look at the typical structure of a Tapestry application, and developed a first page and a couple of simple components. A Tapestry application consists of several pages which can be built from components. A page consists of two artifacts, the page class and an optional template. Components also consist of a Java class and an optional template, and can be nested.

Thanks to the Convention over Configuration principle, Tapestry doesn't require any XML configuration files. Only the root package for the application needs to be specified as a context parameter inside the web deployment descriptor `web.xml`. Starting from the root package Tapestry will search for the necessary classes in the following sub-packages:

- `base` - Sub-package for the base functionality. This can contain super classes for pages and components.
- `components` - Sub-package for the component classes.
- `entities`: Sub-package for Hibernate entities (see chapter XREF `ch-hibernate-spring`).
- `mixins`: Sub-package for mixins (see chapter XREF `ch-mixins`).
- `pages`: Sub-package for page classes.
- `services`: Sub-package for the application module class. The services are also located in this package.

You have seen that creating a component is just a matter of writing a simple Java class and eventually a template, and storing them in a proper package. Tapestry does the rest and resolves a tag name to a proper component class.

Tapestry IoC allows pages and components to be POJOs. The configuration of the IoC container is based on a couple of naming conventions. The name of the application module matches the name of the `TapestryFilter`, as defined in

web.xml, with the suffix Module.

Finally, the Model-View-Controller (MVC) pattern is introduced and the way this is applied in Tapestry. You have learned that pages and components act as controllers. The controller logic is implemented inside event handler methods. These methods handle events which are triggered by Tapestry. Events are discussed in chapter XREF ch-quickstart in detail. You have also learned that, contrary to many MVC frameworks, Tapestry does not depend on JSP technology. Tapestry uses templates as view.

Let's move on to chapter XREF ch-quickstart to dive into the details on the Tapestry Markup Language.