Tapestry 5 Tutorial

by Howard M. Lewis Ship

Table of Contents

Introduction

What is Tapestry?

Welcome to Tapestry!

This is a tutorial for people who will be creating Tapestry 5 applications. It doesn't matter whether you have experience with Tapestry 4 (or Tapestry 3, for that matter) or whether you are completely new to Tapestry. In fact, in some ways, the less you know about web development in general, and Tapestry in particular, the better of you may be ... that much less to unlearn!

You do need to have a reasonable understanding of HTML, a smattering of XML, and a good understanding of basic Java language features, and a few newer things such as Java Annotations.

If you look on the Tapestry web site, you'll see the following description of Tapestry:

> "Tapestry is an open-source framework for creating dynamic, robust, highly scalable web applications in Java. Tapestry complements and builds upon the standard Java Servlet API, and so it works in any servlet container or application server.
>
> Tapestry divides a web application into a set of pages, each constructed from components. This provides a consistent structure, allowing the Tapestry framework to assume responsibility for key concerns such as URL construction and dispatch, persistent state storage on the client or on the server, user input validation, localization/internationalization, and exception reporting. Developing Tapestry applications involves creating HTML templates using plain HTML, and combining the templates with small amounts of Java code. In Tapestry, you create your application in terms of objects, and the methods and properties of those objects -- and specifically not in terms of URLs and query parameters. Tapestry brings true object oriented development to Java web applications.
>
> Tapestry is specifically designed to make creating new components very easy, as this is a routine approach when building applications.

> Tapestry is architected to scale from tiny applications all the way up to massive applications consisting of hundreds of individual pages, developed by large, diverse teams. Tapestry easily integrates with any kind of back-end, including J2EE, HiveMind and Spring. ❞

What does all that mean?

Let's summarize as this: **you do less, Tapestry does more**.

If you're used to developing web applications using servlets and JSPs, or with Struts, you are simply used to a lot of pain. So much pain, you may not even understand the dire situation you are in! These are environments with no safety net; Struts and the Servlet API has no idea how your application is structured, or how the different pieces fit together. Any URL can be an *action* and any action can forward to any *view* (usually a JSP) to provide an HTML response to the web browser. The pain is the unending series of small, yet important, decisions you have to make as a developer (and communicate to the rest of your team). What are the naming conventions for actions, for pages, for attributes stored in the HttpSession or HttpServletRequest?

Worse yet, the traditional approaches thrust something most unwanted in your face: multi-threaded coding. Remember back to *Object Oriented Programming 101* where an object was defined as a bundle of data and operations on that data? You have to unlearn that lesson as soon as you build a web application, because web applications are multi-threaded. An application server could be handling dozens or hundreds of requests from individual users, each in their own thread, and each sharing *the exact same objects*. Suddenly, you can't store data *inside* an object (a servlet or a Struts Action) because whatever data your store for one user will be instantly overwritten by some other user.

Worse, your objects each have one operation: doGet() or doPost().

Meanwhile, most of your day-to-day work involves deciding how to package up some data already inside a particular Java object and squeeze that data into a URL's query parameters, so that you can write *more* code to convert it back if the user clicks that particular link. And don't forget editing a bunch of XML files to keep the servlet container, or the Struts framework, aware of these decisions.

Just for laughs, remember that you have to rebuild, redeploy and restart your application after virtually any change. Is any of this familiar? Then perhaps you'd appreciate something a little *less* familiar: Tapestry.

Tapestry uses a very different model: a structured, organized world of pages, and components within pages. Everything has a very specific name (that you provide). Once you know the name of a page, you know the location of the Java class for that page, the location of the template for that page, and the total structure of the page. Tapestry knows all this as well, and can make things *just work*.

As well see in the following chapters, Tapestry lets you code in terms of *your* objects. You'll barely see any Tapestry classes, outside of a few Java annotations. 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!

Tapestry also shields you from the multi-threaded aspects of web application development. Tapestry manages the life-cycles of your page and components objects, reserving particular objects to particular threads so that you never have to think twice about threading issues.

Tapestry began in January 2000, and now represents over *seven years* of experience: not just my experience, or that of the other Tapestry committers, but the experience of the entire Tapestry community. Tapestry brings to the table all that experience about the best ways to build scalable, maintainable, robust, internationalized (and more recently) Ajax-enabled applications. Tapestry 5 represents a completely new code base designed to simplify the Tapestry coding model while at the same time, extending the power of Tapestry and improving performance.

So, please read on. Let's go build some web applications the *right* way, the Tapestry way.

About this Book

Program listings and snippets use a `fixed point font`.

HTML and Java tend to be verbose, which makes the boundaries of these pages feel a bit cramped; on occasion a ↵ symbol will be used to indicate an artificial line break that would not exist in the original document. For example:

```
mvn archetype:create ↵
  -DarchetypeGroupId=org.apache.tapestry ↵
  -DarchetypeArtifactId=tapestry-simple ↵
  -DgroupId=org.example ↵
  -DartifactId=hilo ↵
  -DpackageName=org.example.hilo ↵
  -DarchetypeVersion=5.0.2
```

This is one very, very long command and it is entered on a single line.

About the Author

Howard Lewis Ship is the creator of Tapestry, and the Chair of the Apache Tapestry Project Management Committee at Apache. Howard is an independent software consultant, specializing in customized Tapestry training, mentoring, architectural review and project work. Howard lives in Portland, Oregon with his wife Suzanne, a novelist.

Chapter 1

Setting up your Environment

As much as I would like to dive into Tapestry right now, we must first talk about your development environment. The joy and the pain of Java development is the volume of choice available. There's just a bewildering number of JDKs, IDEs and other TLA[1]s out there.

Let's talk about a stack of tools, all open source and freely available, that you'll need to setup. Likely you have some of these, or some version of these, already on your development machine.

JDK 1.5

Tapestry 5 makes use of features of JDK 1.5. This includes Java Annotations, and a little bit of Java Generics.

Eclipse 3.2

Since we're emphasizing a *free and open source* stack, we'll concentrate on the best free IDE.

XMLBuddy

A free and reasonably powerful XML editor that will be useful for editing Tapestry component templates.

XMLBuddy is a product of Bocaloco Software (http://xmlbuddy.com/) and comes in both free and commercial editions. Installation directions are available on the site.

XMLBuddy is just a suggestion, you are free to use whatever XML editor suits your needs, or a plain text editor if your are comfortable with that.

---

[1] Three letter acronyms.

Jetty 5.1

Jetty is an open source servlet container created by Greg Wilkins of Webtide (which offers commercial support for Jetty). Jetty is high performance and designed for easy embedding in other software. We choose the 5.1 release, rather than the cutting edge Jetty 6, because it is compatible with Jetty Launcher (see below).

You can find out more about Jetty from its home page: http://mortbay.org.

You can download Jetty from http://docs.codehaus.org/display/JETTY/Downloading+and+Installing.

Jetty Launcher

Jetty Launcher is a plugin for Eclipse that makes it easy to launch Jetty applications from within Eclipse. This is a great model, since you can run or debug directly from you workspace without wasting time packaging and deploying.

Jetty Launcher was created by Geoff Longman, and is available from http://jettylauncher.sourceforge.net/. Installation is easy, simply point Eclipse's update manager at http://jettylauncher.sourceforge.net/updates/.

Caution: JettyLauncher is only compatible with Jetty 4 and Jetty 5. It *does not* work with Jetty 6.

Maven

Maven is a software build tool of rather epic ambitions. It has a very sophisticated plugin system that allows it to do virtually anything, though compiling Java code, building WAR and JAR files, and creating reports and web sites are its forte.

Perhaps the biggest advantage of Maven over, say, Ant, is that it can download project dependencies (such as the Tapestry JAR files, and the JAR files Tapestry itself depends on) automatically for you, from one of several central repositories.

We'll be using Maven to set up our Tapestry applications. Maven 2.0.5 is available from http://maven.apache.org/download.html.

Maven Plugin

The Maven Plugin for Eclipse integrates Maven and Eclipse. It includes some features for editting the pom.xml (the Maven project description file which identifies, among many other things, what JAR files are needed by the project). More importantly, a Maven-enabled project automatically stays synchronized with the POM, automatically linking Eclipse project classpath to files from the local Maven repository.

The plugin is available by pointing the Eclipse update manager at http://m2eclipse.codehaus.org/update/. Make sure to use version **0.0.9** (newer versions have had stability issues).

Tapestry 5.0.2

You should not have to download this directly; as we'll see, Maven should take care of downloading Tapestry, and its dependencies, as needed.

Caution: this book is being written in parallel with Tapestry 5. In some cases, the screen-shots may not be entirely accurate and the version number for Tapestry is in flux, with snapshot releases occuring frequently, and new dot releases every few weeks.

Chapter 2

Creating Your First Tapestry Project

Before we can get down to the fun, we have to create an empty application. Tapestry uses a feature of Maven to do this: *archetypes* (a too-clever way of saying "project templates").

What we'll do is create an empty shell application using Maven, then import the application into Eclipse to do the rest of the work.

Before proceeding, we have to decide on three things: A Maven *group id* and *artifact id* for our project and a root *package name*. Maven uses the group id and artifact id to provide a unique identity for the application, and Tapestry needs to have a base package name so it knows where to look for pages and components.

Our goal in chapter 3 is to create an application that can play a simple game of Hi/Lo with us (a number guessing game), so calling our application "hilo" is a good start. We'll use the group id **org.example** and the artifact id **hilo** and combine the two for the package name: **org.example.hilo.**

Now we're ready to use Maven to build our package. On the command line, we'll tell Maven which archetype to use (there's lots of archetypes, some built in to Maven, others specific to other projects), and tell it about our particular configurations.

The final command line is:

```
mvn archetype:create ↵
  -DarchetypeGroupId=org.apache.tapestry ↵
  -DarchetypeArtifactId=tapestry-simple ↵
  -DgroupId=org.example ↵
  -DartifactId=hilo ↵
  -DpackageName=org.example.hilo ↵
  -DarchetypeVersion=5.0.2
```

… which is quite a doozy! However, you only have to type this once per project.

Execute that command inside a working directory, and a subdirectory, "hilo", will be created.

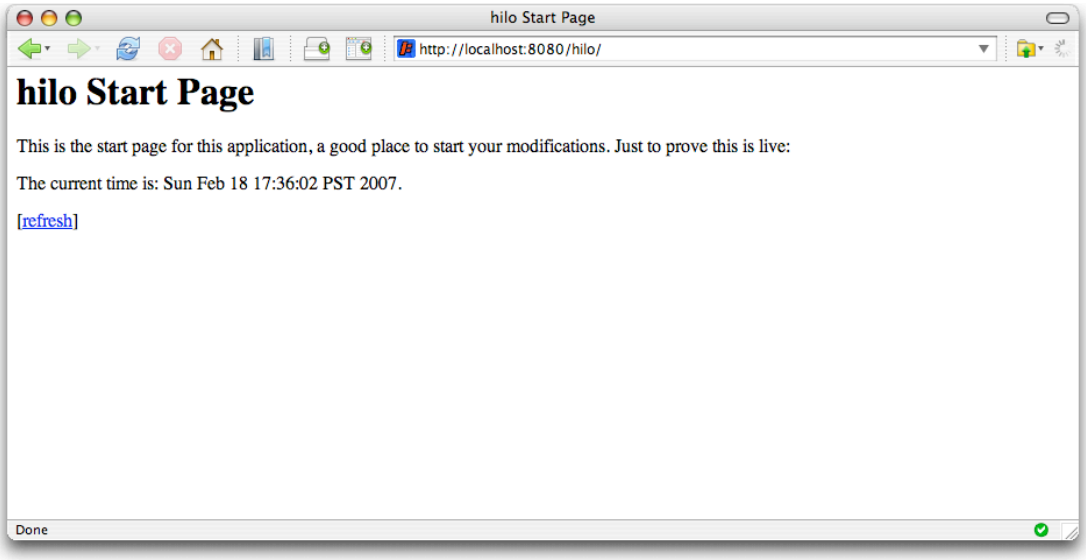
The first time you execute this command, Maven will spend quite a while downloading all kinds of JARs into your local repository, which can take a minute or more. Later, once all that is already available locally, the whole command executes in under a second.

One of the first things you can do is use Maven to run Jetty directly. Change into the new `hilo` directory, and run:

```
mvn jetty:run
```

Again, the first time there's a dizzying number of downloads, but before you know it, the Jetty servlet container is up and running.

You can open a web browser to http://localhost:8080/hilo to see the running application:



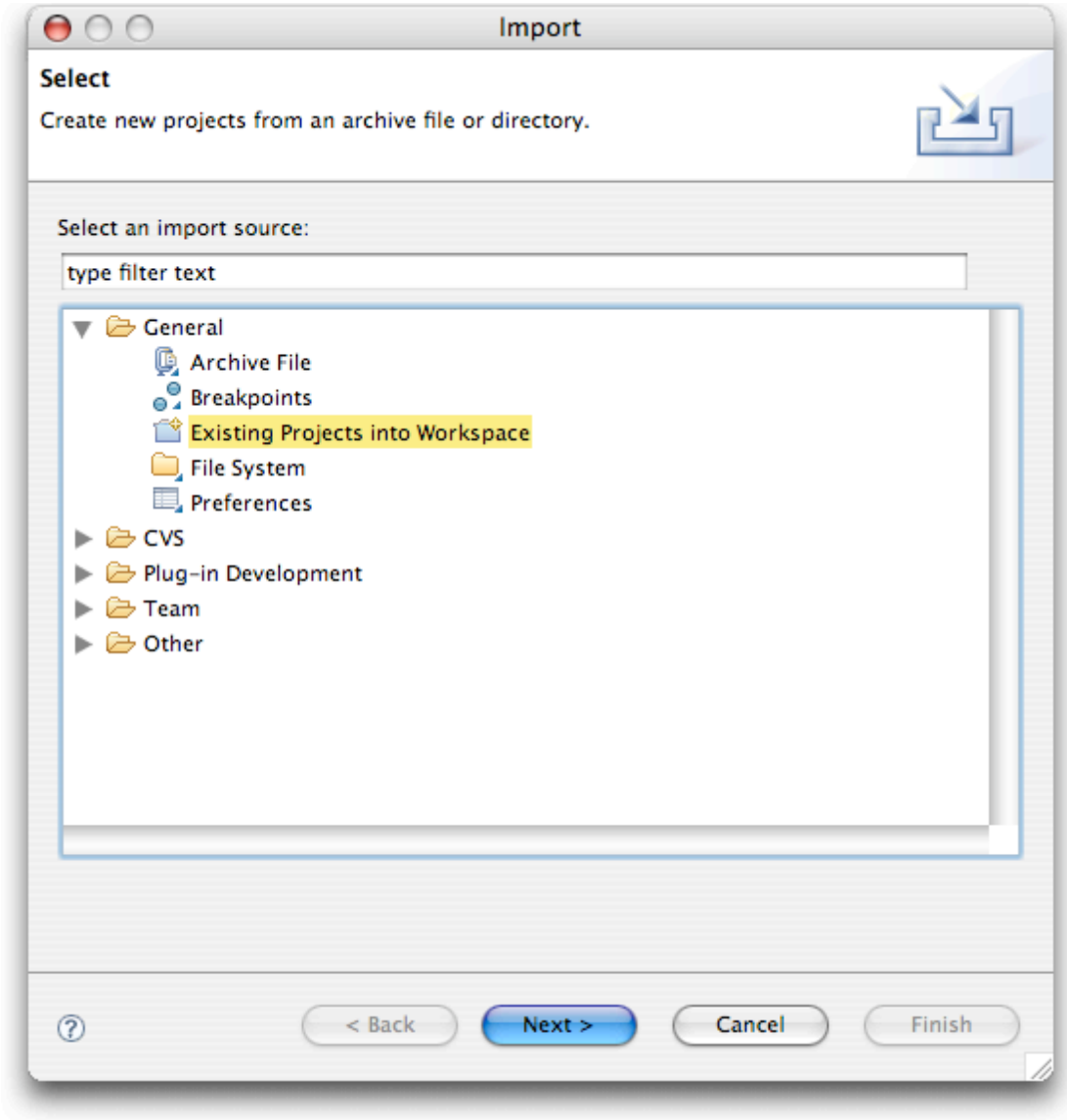
The date time in the middle of the page proves that this is a live application.

Let's look at what Maven has generated for us. To do this, we're going to load the project inside Eclipse and continue from there. Start by hitting Control-C in the Terminal window to close down Jetty.

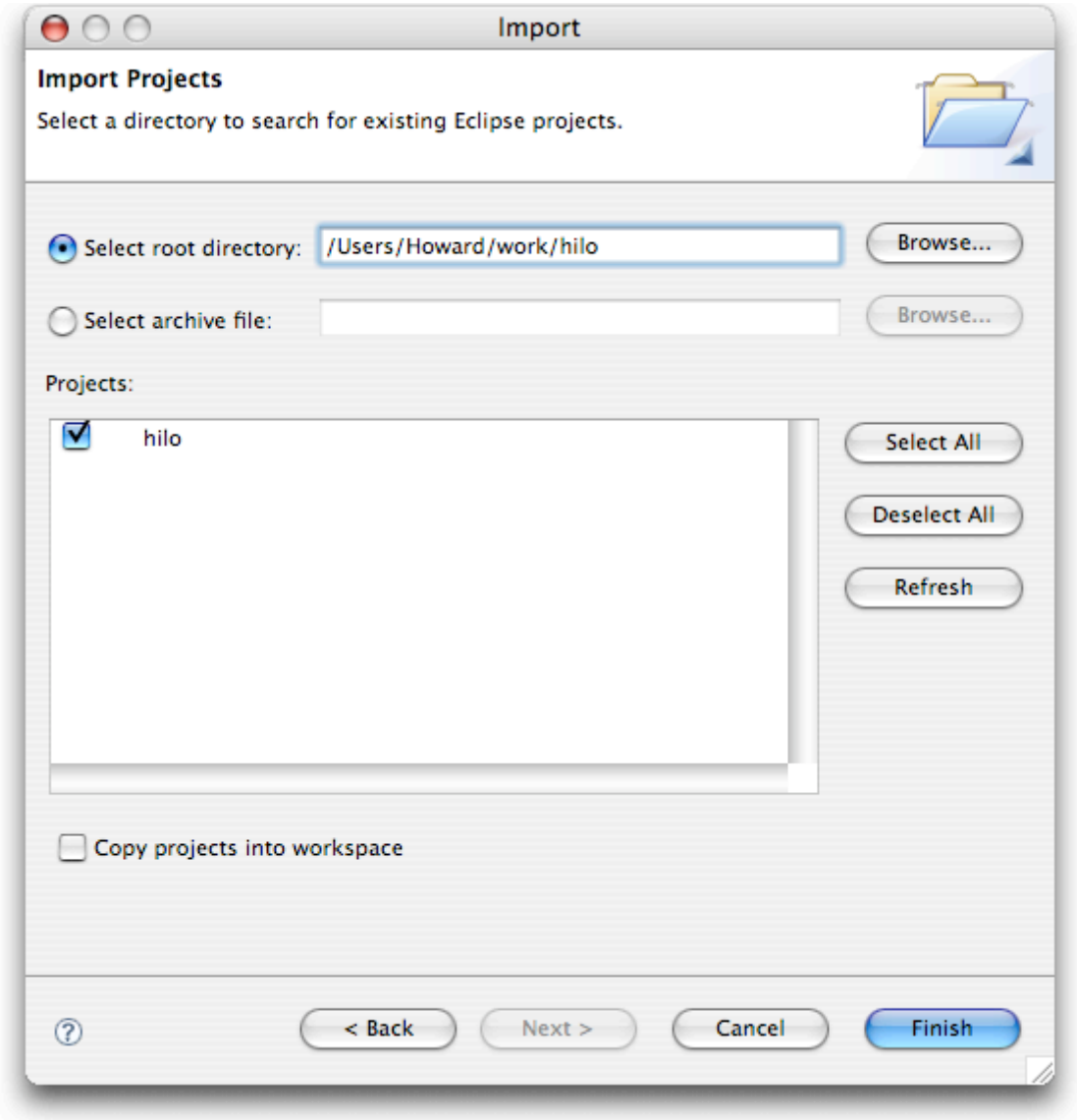
Launch Eclipse and switch over to the Java Browsing Perspective.

Right click inside the Projects view and select **Import …**

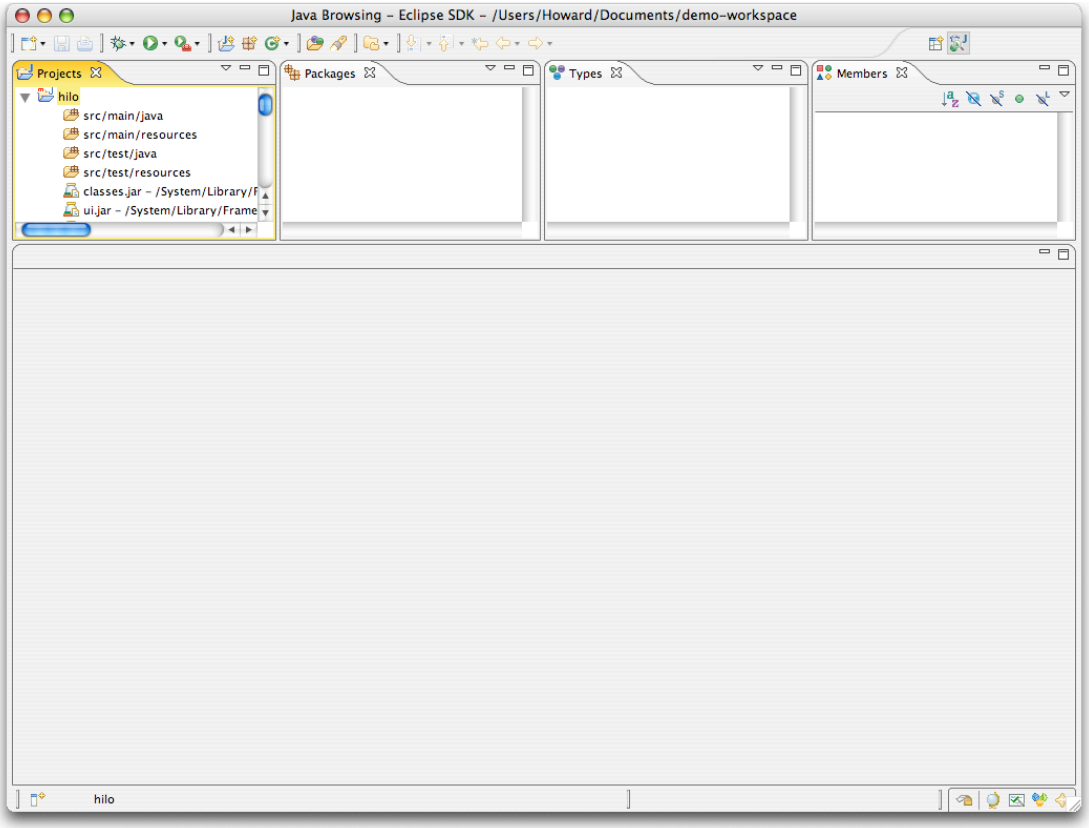
Choose the "existing projects" option:



Now select the folder created by Maven:



When you click Finish, the project will be loaded into the Workspace:



Maven dictates the layout of the project:

- Java source files under src/main/java
- Web application files under src/main/webapp (including src/main/webapp/WEB-INF)
- Java tests sources under src/test/java
- Non-code resources under src/main/resources[2] (and src/test/resources)

The Maven Plugin (inside Eclipse) has found all the referenced libraries in your local Maven repository and compiled the two classes created by the tapestry-simple archetype.

Let's look at what the archetype has created for us, starting with the web.xml file, which is stored as src/main/webapp/WEB-INF/web.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app
```

[2] As we'll see, Tapestry uses a number of non-code resources, such as template files and message catalogs, which will ultimately be packaged into the WAR file.

```
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>hilo Tapestry 5 Application</display-name>
    <context-param>
        <param-name>tapestry.app-package</param-name>
        <param-value>org.example.hilo</param-value>
    </context-param>
    <filter>
        <filter-name>app</filter-name>
        <filter-class>org.apache.tapestry.TapestryFilter ↵
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>app</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

This is short and sweet: you can see that the package name you provided earlier shows up as the tapestry.app-package context parameter; the TapestryFilter instance will use this information to locate the Java classes we'll look at next.

Tapestry 5 operates as a *servlet filter* rather than as a traditional *servlet*. In this way, Tapestry has a chance to intercept all the incoming requests to see which ones apply to Tapestry pages or other resources. The net effect is that you don't have to maintain any additional configuration for Tapestry to operate, no matter how many pages or components you add to your application.

Tapestry has a special case for a URL that specifies the host and the context ("/hilo" in this case) but nothing else … it renders the Start page of the application. In this case, Start is the only page in the application. Let's see what it looks like.

Tapestry pages minimally consist of an ordinary Java class plus a component template file.

Let's start with the template, which is stored as src/main/webapp/WEB-INF/Start.html. Tapestry component templates are well formed XML documents. This means you can use any available XML editor. Templates may even have a DOCTYPE or an XML schema to validate the structure of the template[3]. For the most part, the template looks like ordinary XHTML:

```
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">
    <head>
        <title>hilo Start Page</title>
```

[3] That is, your build process may use a tool to validate your templates. At runtime, when Tapestry reads the template, it does not use a validating parser.

```
    </head>
    <body>
        <h1>hilo Start Page</h1>

        <p> This is the start page for this application, ↵
a good place to start your modifications.
            Just to prove this is live: </p>

        <p> The current time is: ${currentTime}. </p>

        <p>
            [<a t:type="PageLink" t:page="Start">refresh</a>]
        </p>
    </body>
</html>
```

The goal in Tapestry is for component templates, such as Start.html, to look as much as possibly, like ordinary, static HTML files. In fact, the expectation is that in many cases, the templates *will* start as static HTML files, created by a web developer and then be *instrumented* to act as live Tapestry pages.

Tapestry hides non-standard elements and attributes inside the namespace. By convention, the prefix "t:" is used for this namespace, but that is not required.

The xmlns:t attribute on the first line connects the prefix, "t:", to the Tapestry schema. Tapestry's internal template parser recognizes that URL.

There's only two Tapestry items on this page. First is the way we display the current date and time: `${currentTime}`. This syntax is used to access a property of the page object, a property named currentTime. Tapestry calls this an *expansion*. The value inside the braces is the name of a property supplied by the page. As we'll see in a bit, this is just the tip of the iceberg for expansions.

The other item is the link used to refresh the page. The type, "PageLink", is the name of a built-in Tapestry component. PageLink's job is to create a link to another page within the application. In this case, it will create a link back to the Start page.

The URL that the PageLink component will generate is http://localhost:8080/hilo/start. Tapestry is case-insensitive (http://localhost:8080/hilo/START would work just as well[4]) and Tapestry generates lower-case URLs because those are more visually pleasing.

Clicking the link sends a request to re-render the page; the template and Java object are re-used to generate the HTML sent to the browser.

---

[4] The servlet container is not so forgiving, and expects an exact match on the context name portion of the URL: "/hilo".

The final piece of this puzzle is the Java class for the page. Tapestry has very specific rules for where page classes go. Remember that package name? Tapestry adds a sub-package, "pages" to it, and the Java class goes there. Thus the full Java class name is org.example.hilo.pages.Start, stored under the src/main/java folder of the project. Here's that code:

```
package org.example.hilo.pages;

import java.util.Date;

/**
 * Start page of application hilo.
 *
 */
public class Start
{
  public Date getCurrentTime()
  {
    return new Date();
  }
}
```

That's pretty darn simple: No classes to extend, no interfaces to implement, just a very pure POJO (Plain Old Java Bean). You do have to meet the Tapestry framework half-way: you need to put the Java class in the expected package, org.example.hilo.pages, the class must be public, and you need to make sure there's a public no-arguments constructor (here, the Java compiler has done that for us).

The template referenced the property currentTime and we're providing that property, as a *synthetic property*, a property that is computed on the fly (rather than stored in an instance variable).

This means that every time the page renders, a fresh Date instance is created, which is just what we want.

As the page renders, it generates the markup that is sent to the client. For most of the page, that markup is exactly what came out of the component template: this is called the *static content* (we're using the term "static" to mean "unchanging"). The expansion, `${current-Time}`, is *dynamic*: different every time. Tapestry will read that property and convert the result into a string, and that string is mixed into the stream of markup sent to the client[5].

---

[5] We'll often talk about the "client" and we don't mean the people you send your invoices to: we're talking about the client web browser. Of course, in a world of web spiders and screen scrapers, there's no guarantee that the thing on the other end is a web browser. You'll often see low-level HTML and HTTP documentation talk about the "user agent".

Likewise, the PageLink component is dynamic, in that it generates a URL that is (potentially) different every time.

Tapestry follows the rules defined by Sun's JavaBeans specification: a property name of currentTime maps to two methods: getCurrentTime() and setCurrentTime(). If you omit one or the other of those methods, the property is either read only (as here), or write only.

Tapestry does go one step further: it ignores case when matching properties inside the expansion to properties of the page. In the template, we could say `${currentime}` or `${CurrentTime}` or any variation, and Tapestry will *still* invoke the getCurrentTime() method.

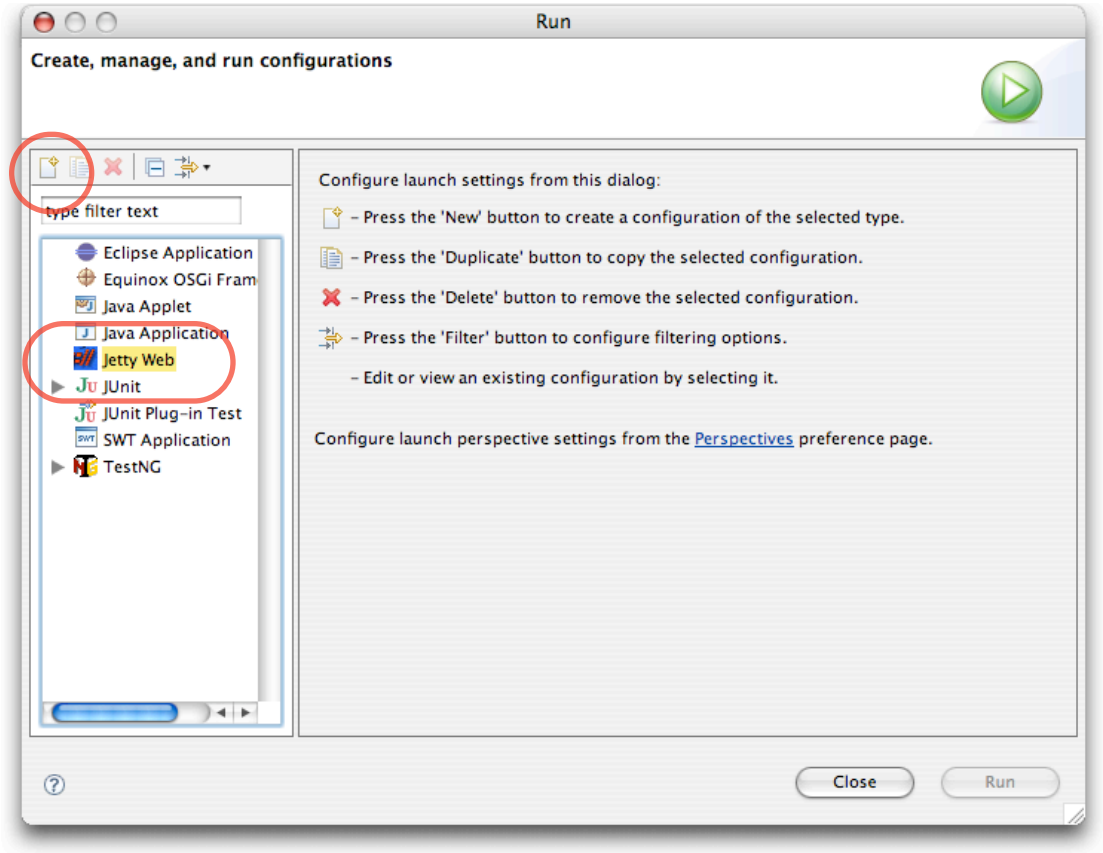
This same case insensitivity applies nearly everywhere in Tapestry; whenever you reference the name of a page, the id of a component, the name of a parameter, or many other things, Tapestry smoothes things out for you, with respect to case. One of the few exceptions is the link between a component class and a component template: the case must match there!

In the next chapter, we're going to implement our Hi/Lo game, but we've got one more task before then, plus a magic trick.

The task is to set up Jetty to run our application directly out of our workspace. This is a great way to develop applications, since we don't want to have to use Maven to compile and run the application … or worse yet, use Maven to package and deploy the application. We want a fast, agile environment that can keep up with our changes and that means we can't wait for redeploys and restarts.

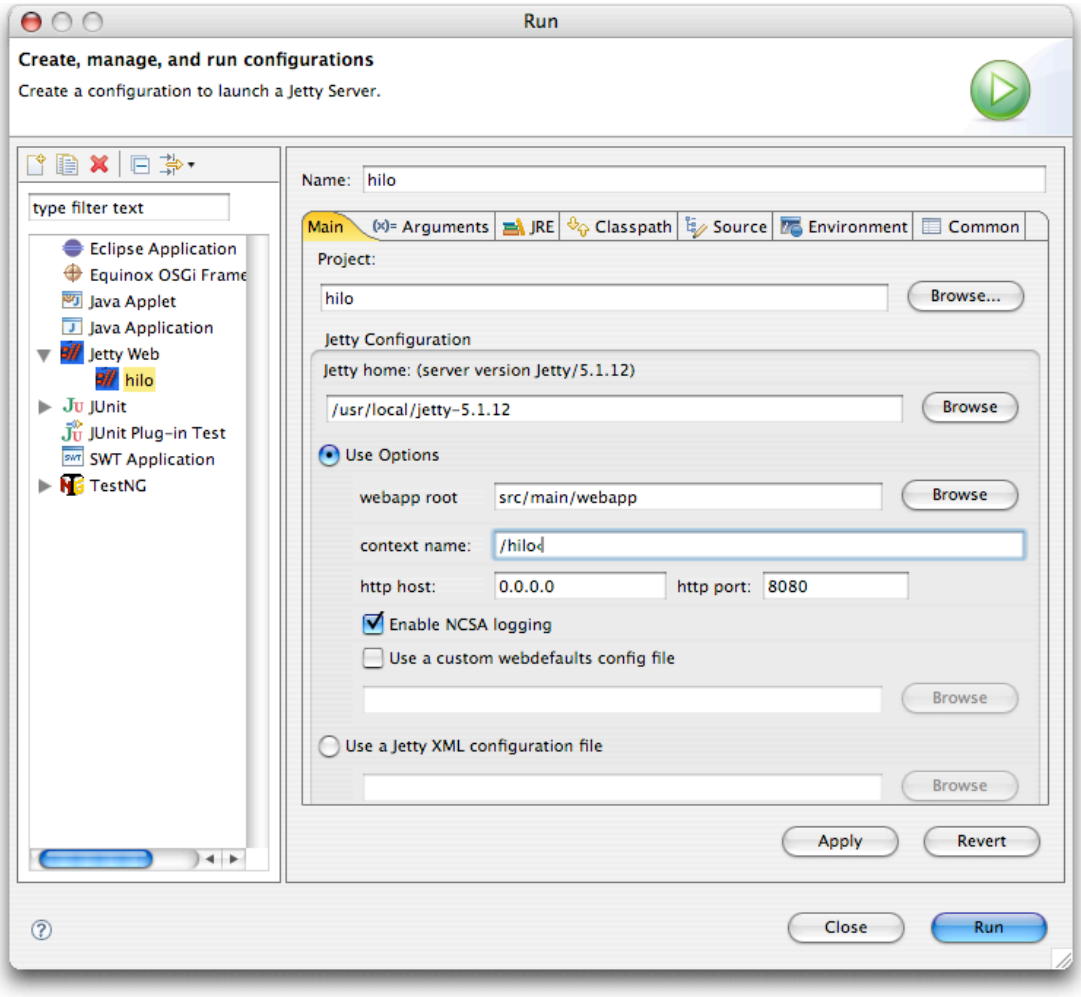
Choose the **Run …** item from Eclipse's **Run** menu to get the launch configurations dialog:

Select Jetty Web and click the **New** button:



Since this is the first time we've used the Jetty launcher, we have to tell it where the Jetty installation directory is.

We'll also set the context name to "/hilo", turn on NCSA logging (always nice to know what requests are coming in), and set the webapp root to src/main/webapp. When you're done, it should look something like:

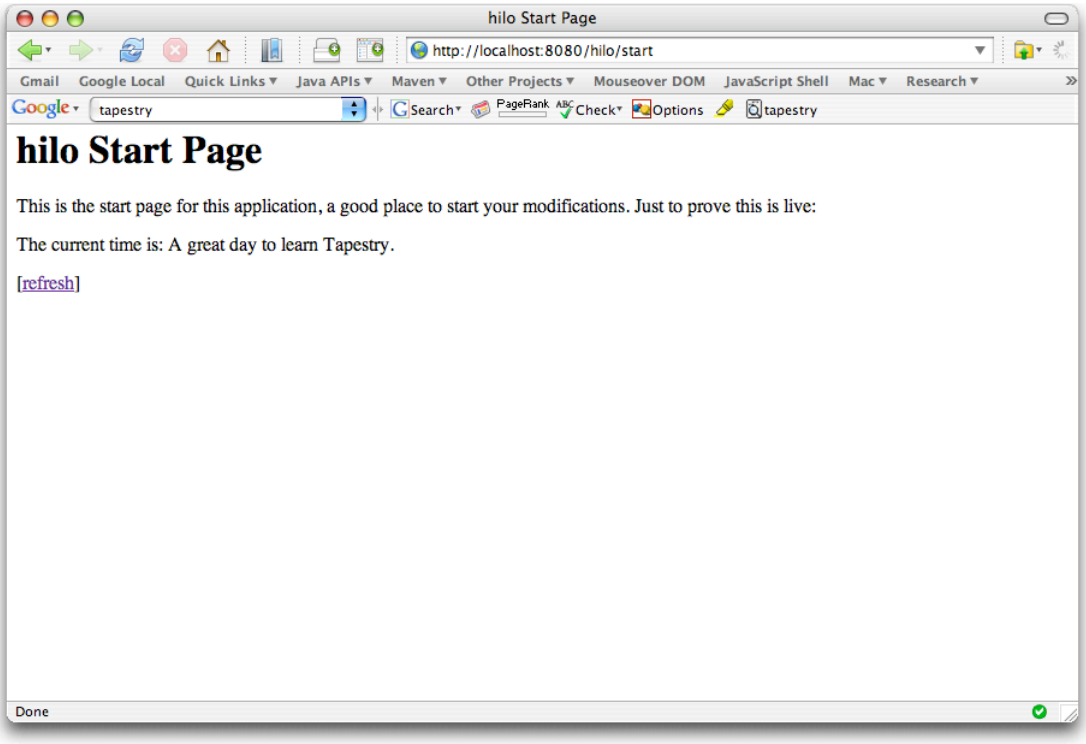


When you click Run, Jetty will start up in the console, and you can jump right into the Start page at http://localhost:8080/hilo/.

Now it's time for the magic trick. Edit Start.java, and change the getCurrentTime() method to:

```
public String getCurrentTime()
{
  return "A great day to learn Tapestry";
}
```

Now click the refresh link in the web browser:



This is one of Tapestry's early *wow factor* features: changes to your component classes are picked up immediately. No restart. No re-deploy. Make the changes and see them *now*. Nothing should slow you down or get in the way of you getting your job done.

Chapter 3

Creating the Hi/Lo Game

Sorry, haven't written this yet!

There's a lot more to come, including a bunch of stuff about forms and input validation, JavaScript, localization, creating new components, and more. I hope what we have here has whetted your appetite … there's a lot of Tapestry to absorb. And the framework itself is evolving at a rapid pace!

Keep monitoring my blog (http://tapestryjava.blogspot.com) for updates concerning this tutorial and Tapestry in general.