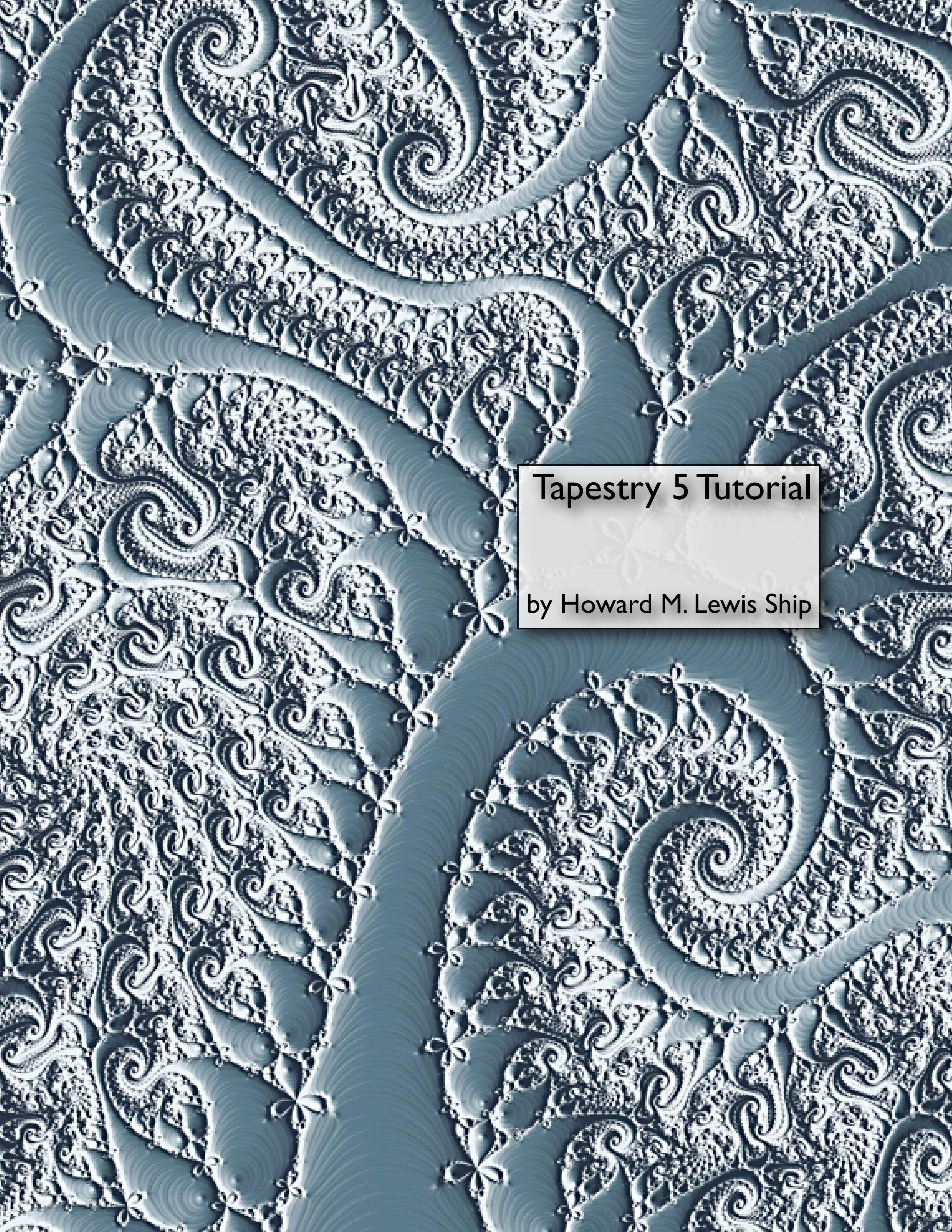


Tapestry 5 Tutorial

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=quickstart ↵
  -DarchetypeVersion=5.0.3 ↵
  -DgroupId=org.example ↵
  -DartifactId=hilo ↵
  -DpackageName=org.example.hilo
```

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 5.0.4

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 occurring 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=quickstart ↩
  -DarchetypeVersion=5.0.4 ↩
  -DgroupId=org.example ↩
  -DartifactId=hilo ↩
  -DpackageName=org.example.hilo
```

… which is quite a doozy! However, you only have to type this once per project (if you're creating a lot of projects, you should create a script or macro to help!).

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

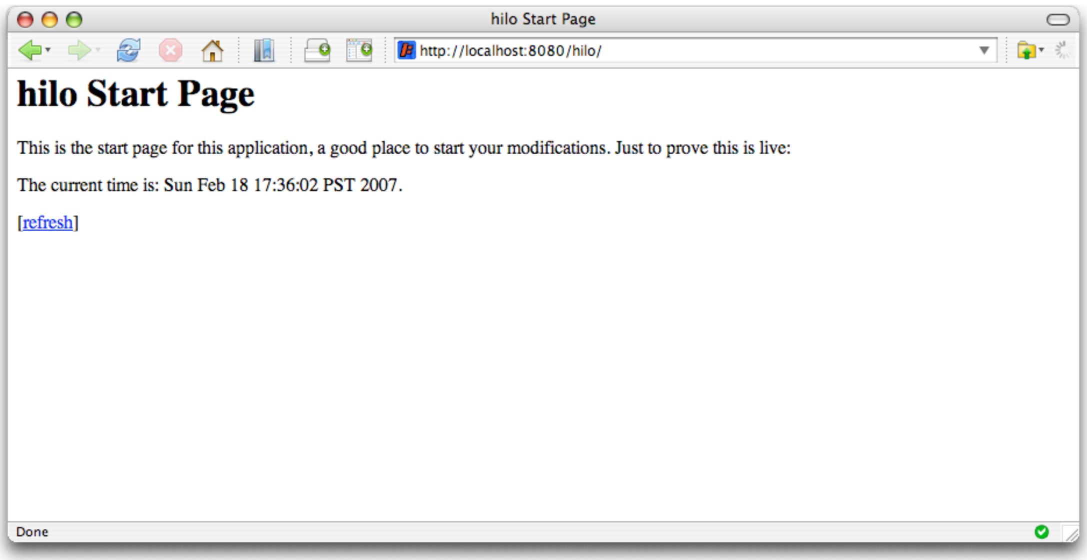
The first time you execute this command, Maven will spend quite a while downloading all kinds of JARs into your local repository, which can take a minute or more. Later, once all that is already available locally, the whole command executes in under a second.

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:



hilo Start Page

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

The current time is: Sun Feb 18 17:36:02 PST 2007.

[refresh]

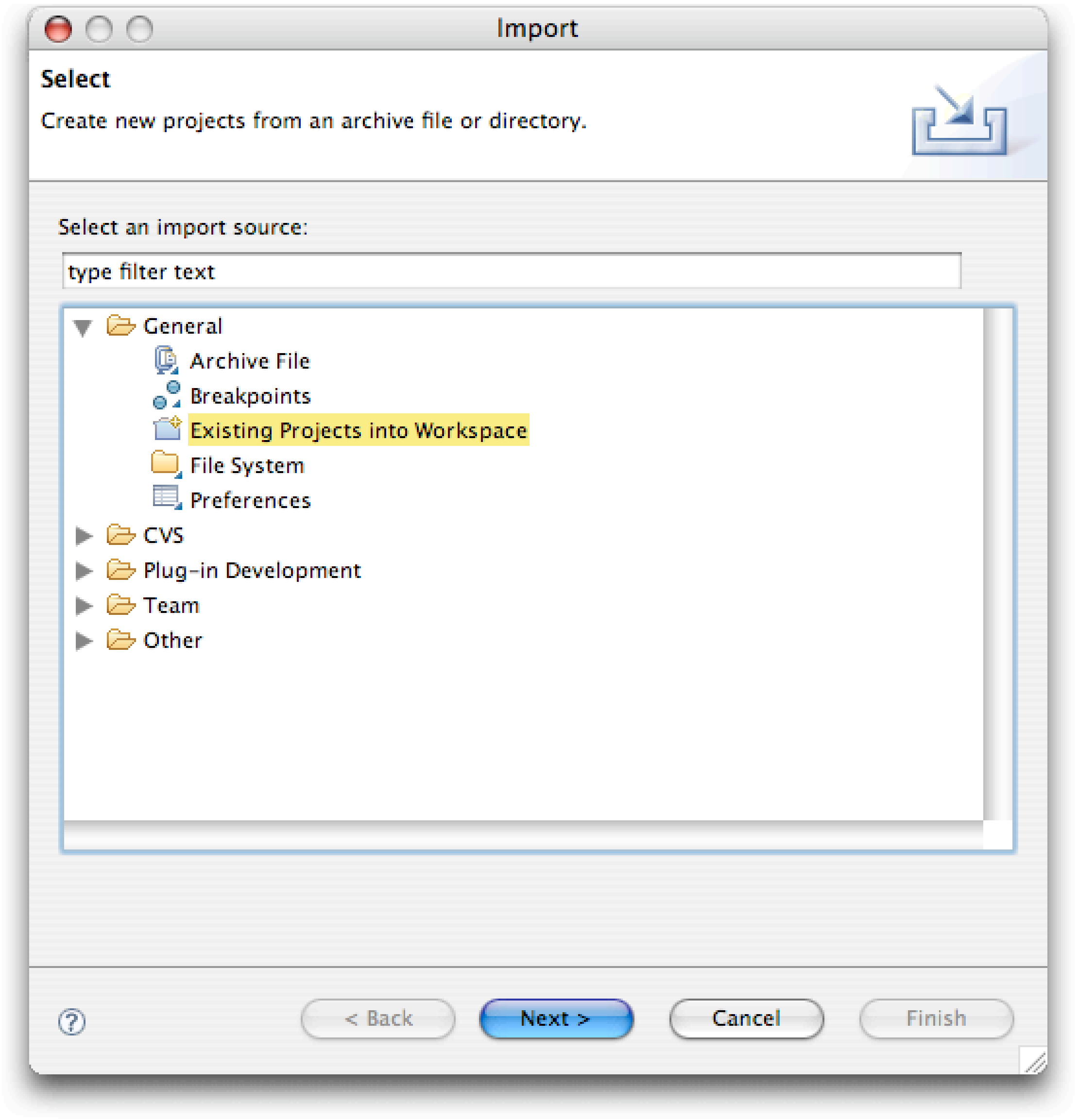
The date time in the middle of the page proves that this is a live application.

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.

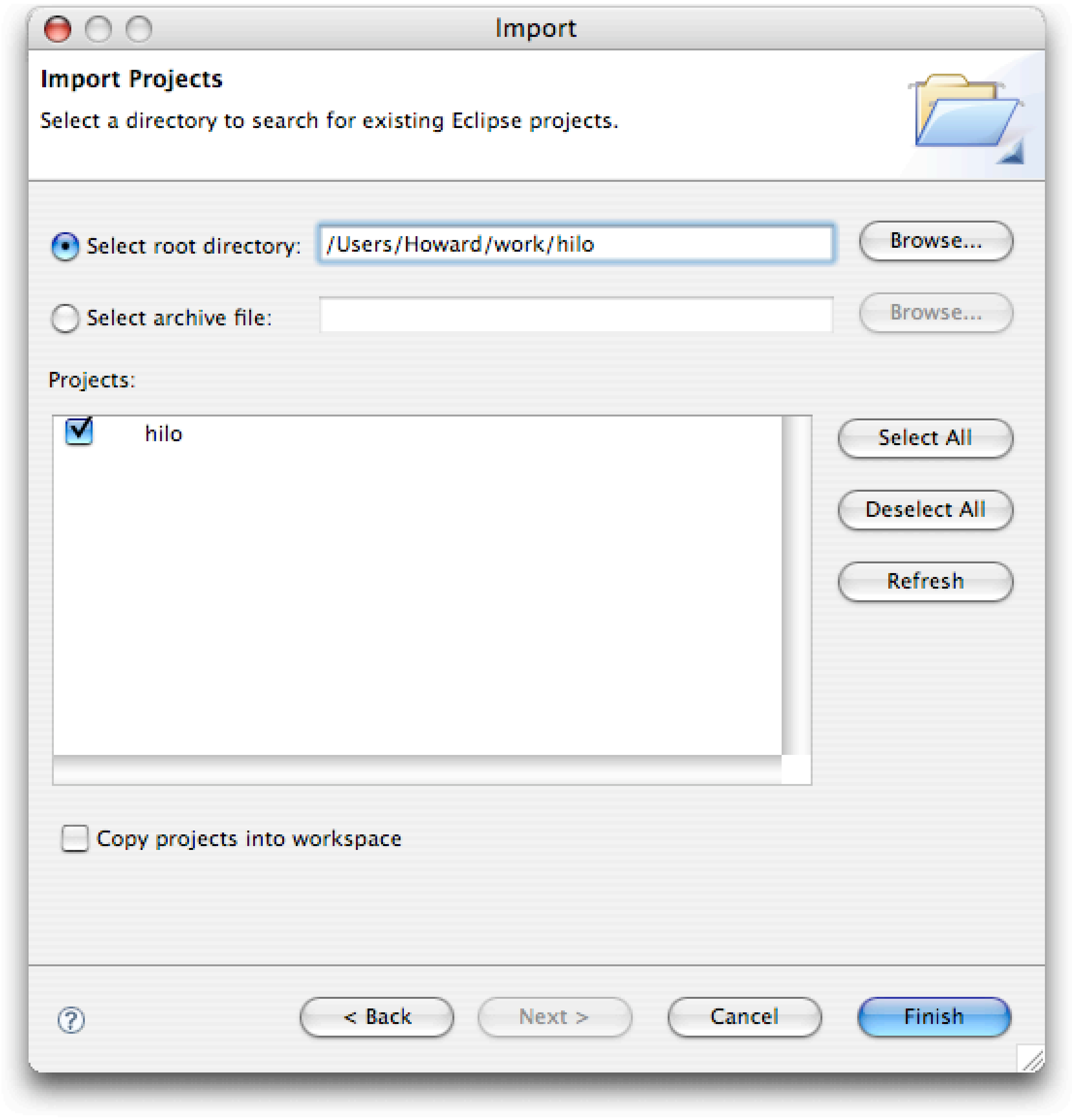
Launch Eclipse and switch over to the Java Browsing Perspective.

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

Choose the "existing projects" option:



Now select the folder created by Maven:



Import

Import Projects

Select a directory to search for existing Eclipse projects.

(●) Select root directory: /Users/Howard/work/hilo

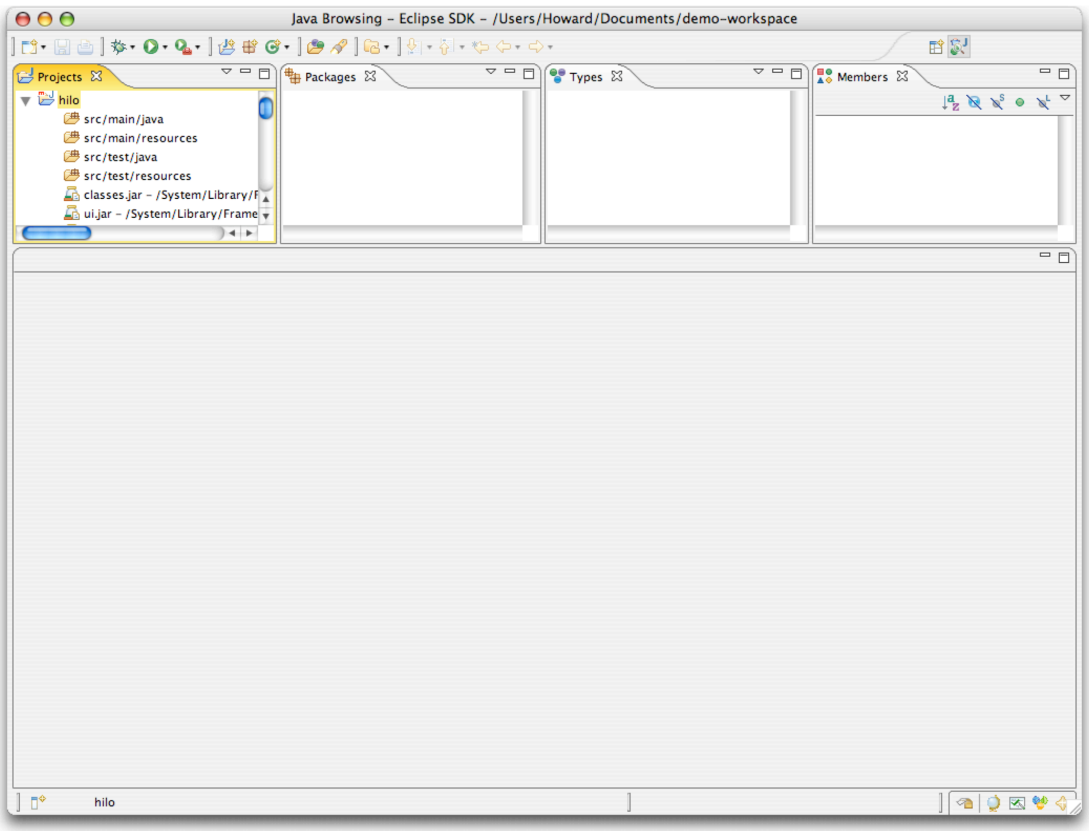
( ) Select archive file:

Projects:

[✓] hilo

[ ] Copy projects into workspace

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


Java Browsing – Eclipse SDK – /Users/Howard/Documents/demo-workspace
Projects
Packages
Types
Members
hilo
src/main/java
src/main/resources
src/test/java
src/test/resources
classes.jar – /System/Library/F
ui.jar – /System/Library/Frame
hilo

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:

```
• 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:

```
• src/main/webapp/WEB-INF/Start.html
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">
    <head>
```

[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.

```
        <title>hilo Start Page</title>
    </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>
            [<t:pagelink page="Start">refresh</t:pagelink>]
        </p>
    </body>
</html>
```

The goal in Tapestry is for component templates, such as Start.html, to look as much as possible like ordinary, static HTML files[4]. 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 constructs 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 later chapters, this is just the tip of the iceberg for what is possible using expansions.

The other item is the link used to refresh the page. We're specifying a component as an XML *element* within the Tapestry namespace. The element name, "pagelink", defines the type of component. PageLink (Tapestry is case insensitive) is a component built into the framework. The attribute, page, is a string – the name of the page to create a link to.

This is how Tapestry works; the Start page contains an *instance* of the PageLink component. The PageLink component is configured via its parameters, which controls what it does and how it behaves.

[4] By "static" we mean unchanging, as opposed to a dynamically generated Tapestry 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[5]) and generates lower-case URLs because those are more visually pleasing.

Tapestry ignores case where ever it can. Inside our template, we configured the PageLink component's page parameter with the name of the page, "Start". Here, too, we could be inexact on case, so use "start" if that works for you[6].

Clicking the link in the web browser sends a request to re-render the page; the template and Java object are re-used to generate the HTML sent to the browser, which results in the updated time showing up in the web browser.

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:

```
• src/main/java/org/example/hilo/pages/Start.html
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 Object). You do have to meet the Tapestry framework half-way:

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

[6] You do have to name your component template file, Start.html, with the same case as your component class (Start). If you get the case wrong, it may work on some operating systems (such as Windows) and not on others (Mac OS X, Linux and most others). This can be really vexing, as it is common to develop on Windows and deploy on Linux or Solaris, so be careful about case in this one area.

- You need to put the Java class in the expected package, org.example.hilo.pages,
- The class must be public
- You need to make sure there's a public no-arguments constructor (here, the Java compiler has quietly provided one 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, `${currentTime}`, 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[7]. 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.

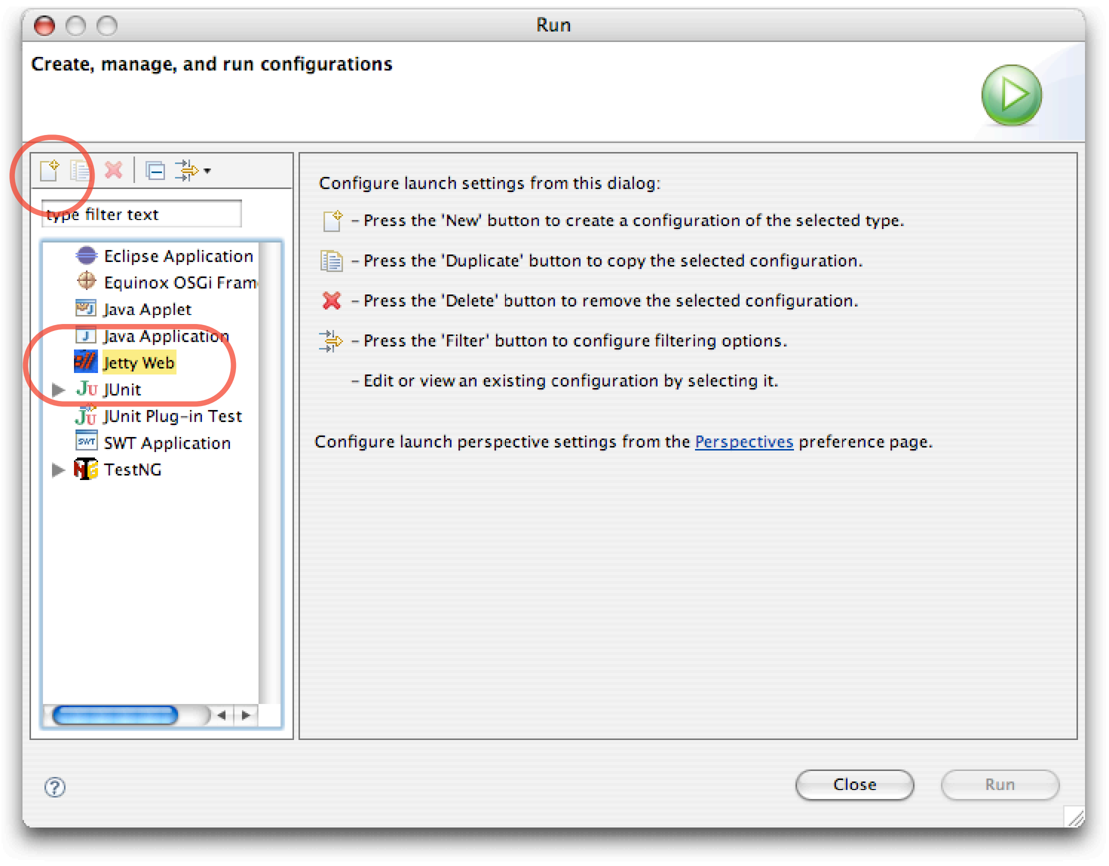
Tapestry does go one step further: it ignores case when matching properties inside the expansion to properties of the page. In the template, we could say `${currentime}` or `${CurrentTime}` or any variation, and Tapestry will *still* invoke the getCurrentTime() method.

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:

[7] 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 really is a web browser. You'll often see low-level HTML and HTTP documentation talk about the "user agent".



Configure launch settings from this dialog:

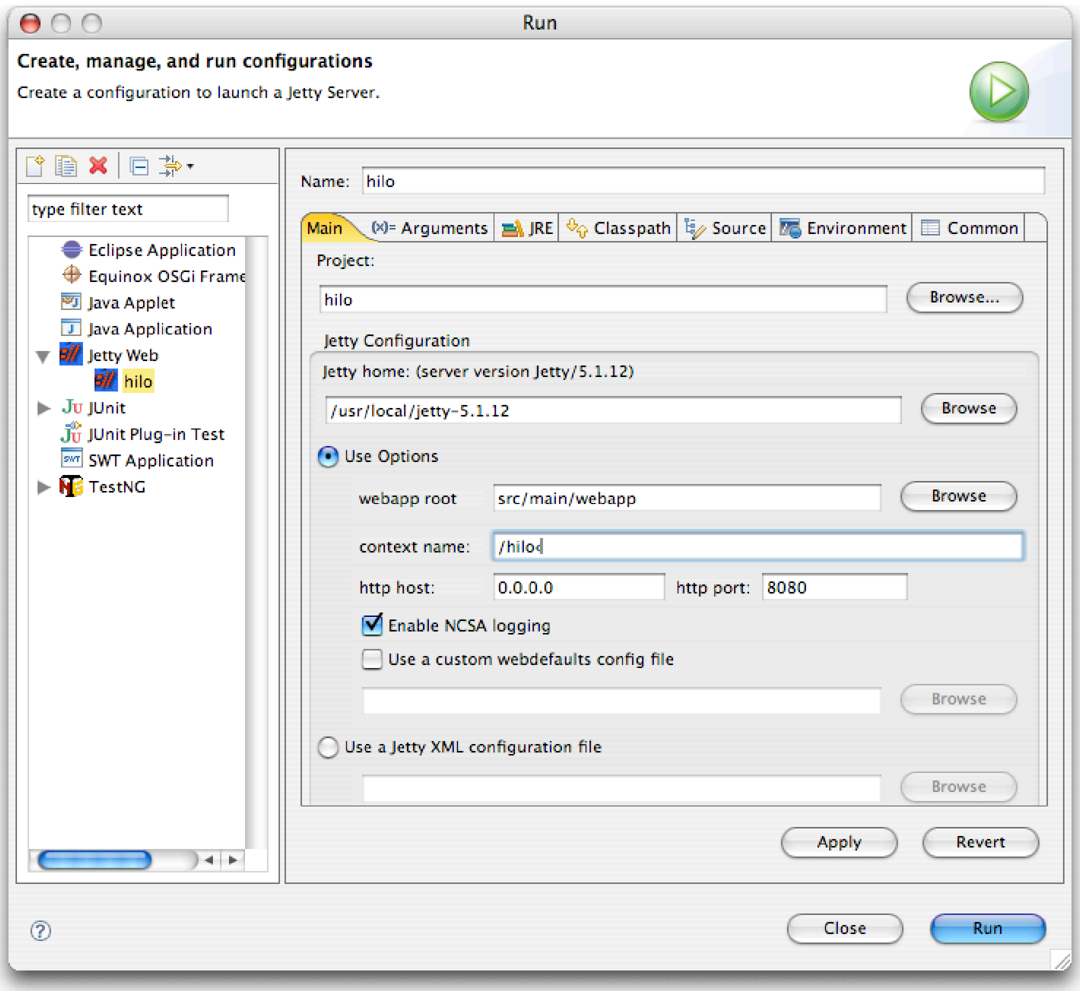
- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
- Press the 'Filter' button to configure filtering options.
- Edit or view an existing configuration by selecting it.

Configure launch perspective settings from the Perspectives preference page.

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:


Run

Create, manage, and run configurations

Create a configuration to launch a Jetty Server.

type filter text

- Eclipse Application
- Equinox OSGi Frame
- Java Applet
- Java Application
- Jetty Web
  - hilo
- JUnit
- JUnit Plug-in Test
- SWT Application
- TestNG

Name: hilo

Main | Arguments | JRE | Classpath | Source | Environment | Common

Project: hilo

Jetty Configuration

Jetty home: (server version Jetty/5.1.12) /usr/local/jetty-5.1.12

[✓ Use Options]

webapp root: src/main/webapp

context name: /hilo

http host: 0.0.0.0 http port: 8080

[✓] Enable NCSA logging

[ ] Use a custom webdefaults config file

[ ] Use a Jetty XML configuration file

Apply Revert

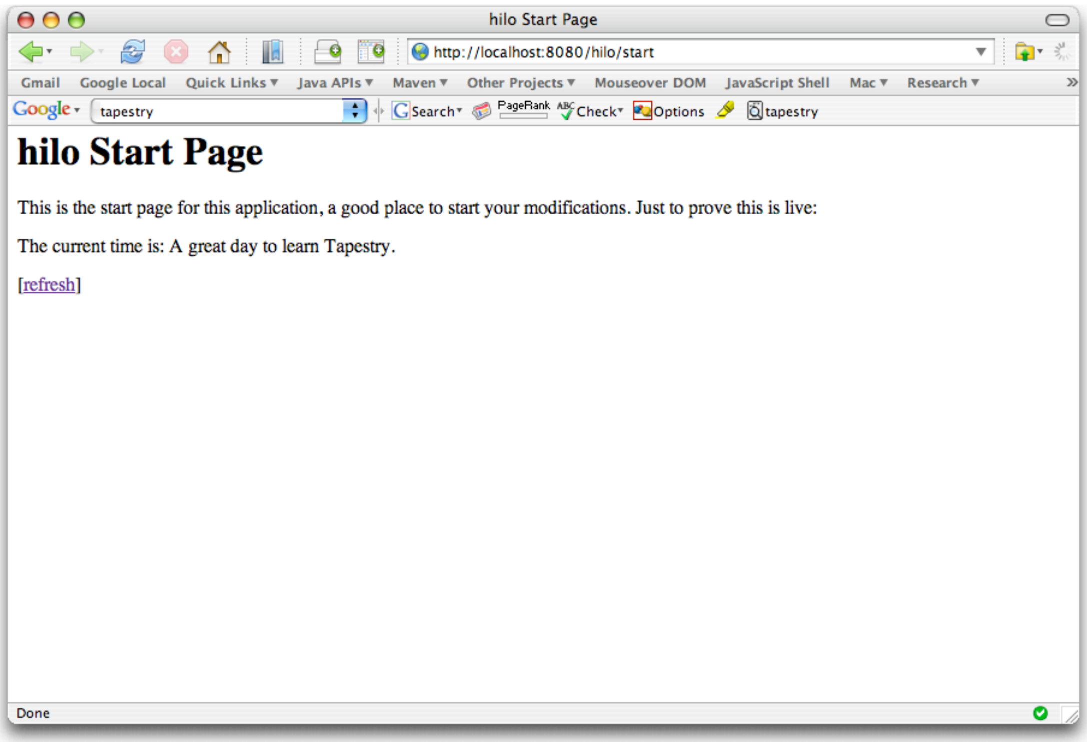
Close Run

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:


hilo Start Page
http://localhost:8080/hilo/start
Gmail Google Local Quick Links Java APIs Maven Other Projects Mouseover DOM JavaScript Shell Mac Research
Google tapestry Search PageRank Check Options tapestry
Done

hilo Start Page

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

The current time is: A great day to learn Tapestry.

[refresh]

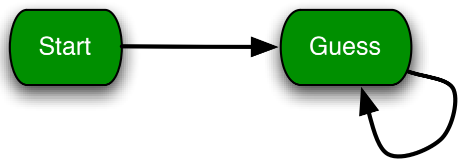
This is one of Tapestry's early *wow factor* features: changes to your component classes are picked up immediately. No restart. No re-deploy. Make the changes and see them *now*. Nothing should slow you down or get in the way of you getting your job done.

Now that we have our basic application set up, and ready to run (or debug) right inside Eclipse, we can start working on our Hi/Lo game in earnest.

Chapter 3

Creating the Hi/Lo Game

Let's start building the Hi/Lo game. We'll build it in small pieces, using the kind of iterative development that Tapestry makes so easy.



Our page flow is very simple: two pages, Start and Guess. The Start page introduces the application and includes a link to start guessing.

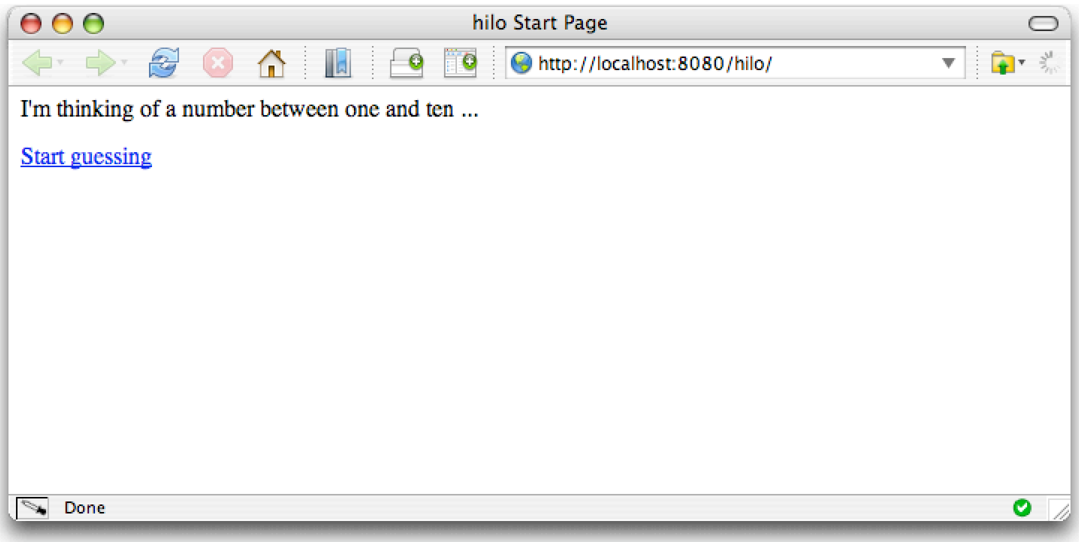
The Guess page will display the user's most recent guess, and provide a series of links to click to guess a number. After each click, the application will respond with a message: "too low", "too high", or "you guessed it!".

Let's start with the Start page's component template:

• src/main/webapp/WEB-INF/Start.html

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

    <p> I'm thinking of a number between one and ten ... </p>

    <p>
      <t:actionlink>Start guessing</t:actionlink>
    </p>
  </body>
</html>
```

We can start up the application right away, and launch the application (http://localhost:8080/hilo):



I'm thinking of a number between one and ten ...

Start guessing

Clicking the link doesn't do anything, however. We haven't told Tapestry what to do when the link gets clicked.

Let's fix that. We'll change the Start class so that it will react when the link is clicked … but what should it do? Well, to start the guessing process, we need to come up with a random number (between one and ten), we need to tell the Guess page about it, and we need to make sure the Guess page is started up to display the response.

• src/main/java/org/example/hilo/pages/Start.java

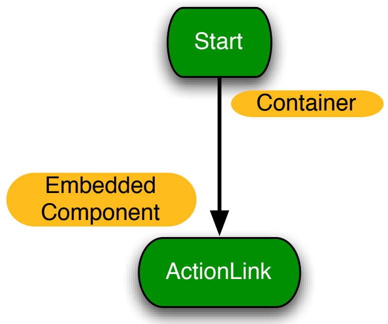
```
package org.example.hilo.pages;

import java.util.Random;

import org.apache.tapestry.annotations.InjectPage;

/**
 * Start page of application hilo.
 */
public class Start
{
  private final Random _random = new Random();

  @InjectPage
  private Guess _guess;
  Object onAction()
  {
    int target = _random.nextInt(9) + 1;

    _guess.setup(target);

    return _guess;
  }
}
```

We'll get to the Guess page in just a moment.

This tiny example demonstrates a lot about Tapestry.

When you click the link, Tapestry invokes the method onAction(). There's a lot of who and why questions tied up in that statement. Let's start with they why.

When you click a link generated by the ActionLink component, a new request is sent to Tapestry. This request locates the component in question and triggers an event within that component.



Tapestry has a structure for the Start page: The ActionLink component is *embedded* inside the Start page. Later we'll see components with their own templates and their own embedded components. In any case, the event is triggered inside the ActionLink component.

The ActionLink component doesn't have any handler for the action event, so the event bubbles up to its container, the Start page.

In the Start page we find a handler, an *event handler method*, for the action event. Tapestry has used a *naming convention* to match the incoming event, "action" to the method onAction(). Later we'll see other variations on this concept, to handle other cases (such as more than one ActionLink component within the page).

If no handler is found, it is not considered an error. The default behavior is just to redisplay the page containing the link. That's why clicking the link before we added some code didn't appear to do anything: Tapestry asked around for someone to handle the event and didn't find anyone, and the page re-rendered so fast it felt like nothing happened at all.

So, because the type of event was "action" (which makes sense, it's generated by an *Action*Link component), and because the Start page had a method named onAction(), Tapestry invoked that method as the handler for the event.

Notice that the method is not public; Tapestry can invoke event handler methods regardless of visibility (they can even be private). Generally, these methods are package private, since that makes it possible to test the methods. Again, that's a subject we'll return to in a later chapter.

Tapestry provides all the scaffolding: recognizing the URL, finding the page and component, and invoking the event handler methods.You get to provide the interesting part, the business logic, inside the event handler method. Here we're generating a random target number for the user to guess. We're also storing it inside the Guess page. The @InjectPage annotation is responsible for connecting the two pages together[8].

Lastly, the method returns the Guess page instance that was just configured. Returning an instance of a page informs Tapestry that the page returned is the page that should render the response, generating the markup that will appear in the user's browser. We'll get back to that in a second.

This scenario is often referred to as the "Tapestry bucket brigade", where logic associated with one page hands off to another page, all in Java code. The benefits of this approach become even more apparent as we start doing more complex things in later chapters.

This code won't compile or execute without a Guess page, so let's create an initial pass at it:

```
• src/main/java/org/example/hilo/pages/Guess.java
ppackage org.example.hilo.pages;

public class Guess
{
  private int _target;

  void setup(int target)
  {
    _target = target;
  }
}
```

The setup() method is package private; it's only intended to be called from the Start page, which is in the same package. Right now, it just stores the target value for later use.

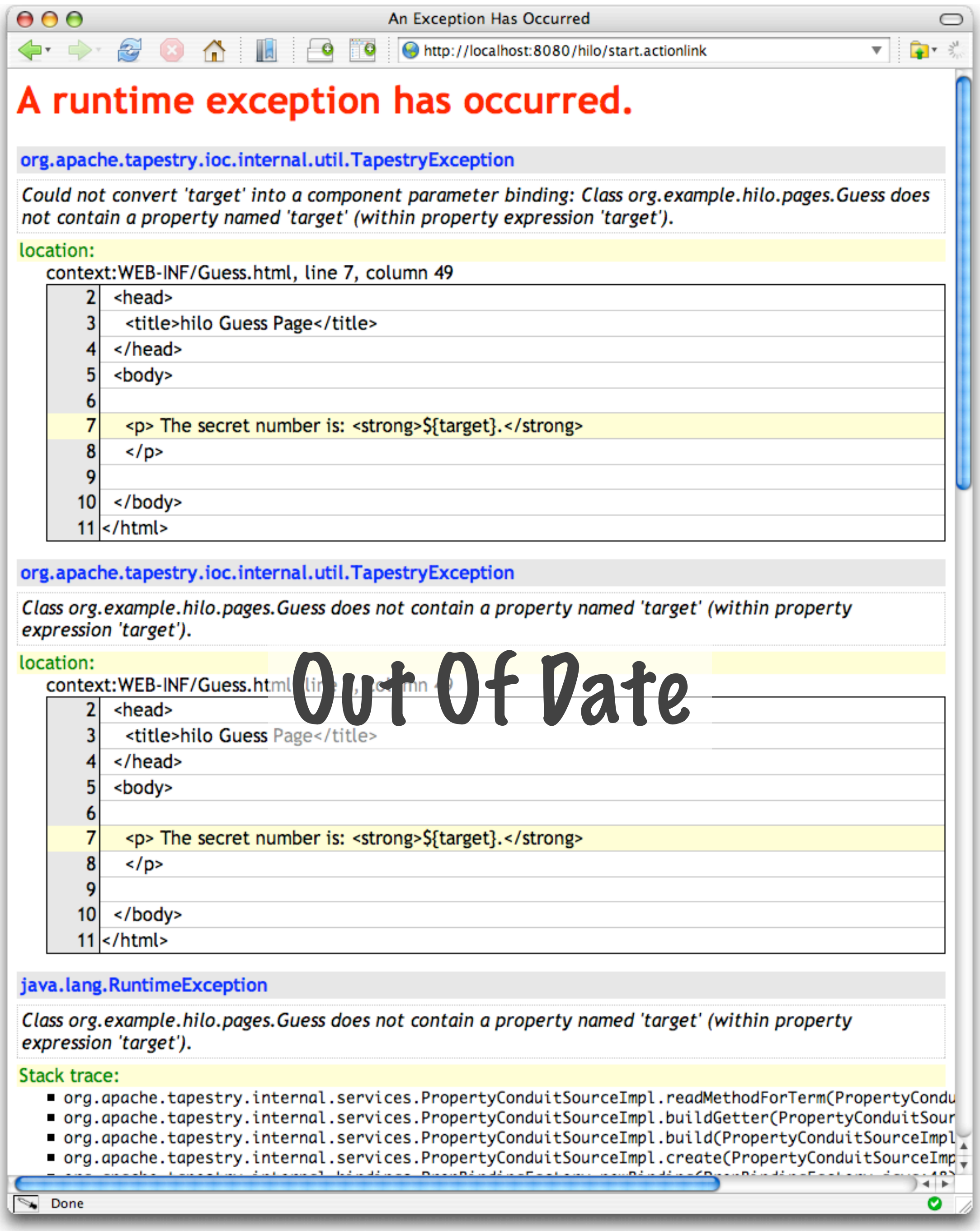
Before we get into the mechanics of displaying numbers for the user to choose, let's just validate that this first step works. We'll create a simple template for the Guess page:

```
• src/main/webapp/WEB-INF/Guess.html
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">
```

[8] There's quite a few annotations in Tapestry and nearly all of them are attached to fields. All fields inside component classes should be private. The annotations only work on private instance fields.

```
  <head>
    <title>hilo Guess Page</title>
  </head>
  <body>

    <p> The secret number is: <strong>${target}.</strong>
    </p>

  </body>
</html>
```

Let's give this a whirl: Click the link on the Start page again and let's see what our first target number is:



An Exception Has Occurred

http://localhost:8080/hilo/start.actionlink

A runtime exception has occurred.

org.apache.tapestry.ioc.internal.util.TapestryException

Could not convert 'target' into a component parameter binding: Class org.example.hilo.pages.Guess does not contain a property named 'target' (within property expression 'target').

location:

context:WEB-INF/Guess.html, line 7, column 49

```
 2  <head>
 3    <title>hilo Guess Page</title>
 4  </head>
 5  <body>
 6
 7    <p> The secret number is: <strong>${target}.</strong>
 8    </p>
 9
10  </body>
11 </html>
```

org.apache.tapestry.ioc.internal.util.TapestryException

Class org.example.hilo.pages.Guess does not contain a property named 'target' (within property expression 'target').

location:

context:WEB-INF/Guess.html, line 7, column 49

```
 2  <head>
 3    <title>hilo Guess Page</title>
 4  </head>
 5  <body>
 6
 7    <p> The secret number is: <strong>${target}.</strong>
 8    </p>
 9
10  </body>
11 </html>
```

java.lang.RuntimeException

Class org.example.hilo.pages.Guess does not contain a property named 'target' (within property expression 'target').

Stack trace:

```
▪ org.apache.tapestry.internal.services.PropertyConduitSourceImpl.readMethodForTerm(PropertyCondu
▪ org.apache.tapestry.internal.services.PropertyConduitSourceImpl.buildGetter(PropertyConduitSour
▪ org.apache.tapestry.internal.services.PropertyConduitSourceImpl.build(PropertyConduitSourceImpl
▪ org.apache.tapestry.internal.services.PropertyConduitSourceImpl.create(PropertyConduitSourceImp
```

Done

Not quite what we were expecting.

The issue here is that the Guess page's template referenced a property named "target" and it doesn't exist (the code only defines the field, not a getter and/or setter method).

Tapestry has a comprehensive exception report page that does a number of useful things:

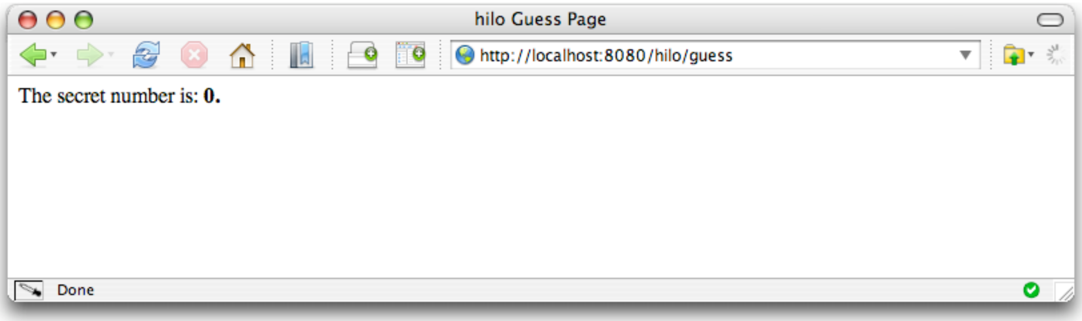
- When the exception is related to a template, it shows you the portion of the template surrounding the error.
- It breaks apart nested exceptions and shows you the whole stack of them.
- It shows the stack trace of just the deepest exception.
- If your scroll down, there are details about the current request (including parameters and headers) and the values stored in the HttpSession.

Of course, you can turn this off in a production application, but while we're assembling our application initially, it's tremendously useful.

Next, we'll add the missing getter method:

```
public int getTarget()
{
  return _target;
}
```

… then hit the browser back button and retry the link. Again, not quite what we expected:



The secret number is: **0.**

No exception this time, but the value is zero. We know that it was at least 1, so what happened?

In Tapestry 5, handling an action request is a two-step process. First the URL for the action request is processed, which invokes the Start page's onAction() method, as outlined above.

The response from that request isn't the HTML markup generated from the Guess page; instead, it's a client-side redirect to the Guess page.

The markup is generated in an *entirely new request*.

Between requests, Tapestry does a lot of housekeeping on the pages. All their normal instance fields, such as the _target field in the Guess class, get reset to their default values.

Why does Tapestry do this?

Well, it gets into a larger question of what life is like for a web application (of any type, in any language) that's deployed and open for the public.

In a deployed application, there will be hundreds or thousands of concurrent users. Think about all the people hitting Amazon or eBay right now; for the largest applications, millions of users isn't out of the question.

Now even for the busiest application with the most frenetic users, most of those users are in "think time". They're reading the page and moving the mouse to the button or link they want to click.

For any given instant, inside the application server you will have requests from dozens or hundreds of users who, from all over the world, have simultaneously clicked their link or button. All of these requests are processed in parallel.

Using traditional Java servlets, all of those users who clicked the same link would be routed through a single, shared servlet instance. Just one object, handling all requests for all users. The first lesson there is that you can't use instance variables, since any data you wrote into an instance variable for safe keeping would be overwritten almost instantly by another thread.

So, in traditional servlet development, you break out the HttpServletRequest and HttpSession objects and store your data as named attributes. Values you just need for the current request go in the HttpServletRequest. Values that need to persist from request to request go in the HttpSesson. Really long lived data goes in a database.

Returning to our initial question ("why are instance variables cleaned out"?) you'll notice that we *are* using instance variables. Tapestry works really hard to lift you out of the painful world of servlet and multithreading and back into the more natural world of objects. Tapestry associates an *instance* of the Start page with each request. In a loaded application, you'll have multiple threads, each handling its own request, each with its own instance of the Start page.

Creating pages the first time is expensive. It's so fast you won't even notice it on your development box, running the application just for you … but in a production server, creating a fresh instance of each page just for one request could add hundreds of milliseconds to each request.

Instead, Tapestry *pools* pages. On each request, the associated page is fetched from the pool. If the pool doesn't have a copy handy, a fresh instance is created. At the end of the request, the page is scrubbed and put back in the pool for later use. Most likely, it won't be reused for a request from the same user, and that's OK … we've scrubbed out any values that may be specific to the first user.

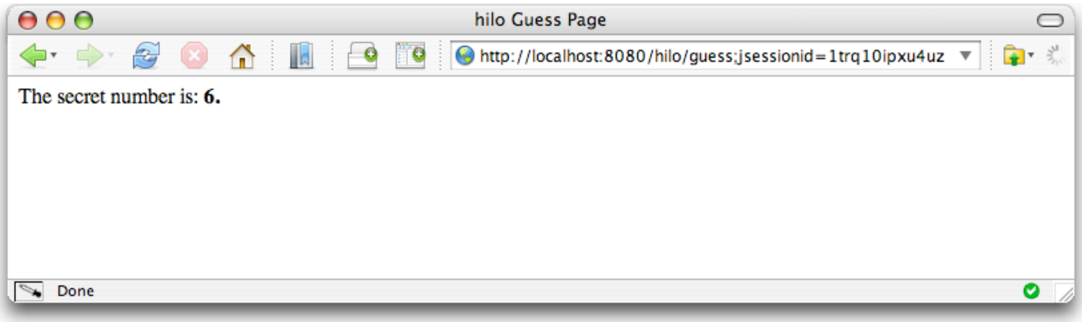
This is your have-your-cake-and-eat-it-too scenario: your code doesn't have to deal with multi-threading issues, but it still runs nice and quick.

So, if we want the target value to show up correctly, we need to make it persist from one request to the next. If you've done work with Java servlets before, you might be asking "how do I get access to the HttpSession?". Well, you can … but you don't want to. All you need to do is change your field a smidgen:

```
@Persist
private int _target;
```

That's the org.apache.tapestry.annotations.Persist annotation, and adding it to a field tells Tapestry that the field's value should be stored in the session between requests. Tapestry takes care of shuttling the value between the field and the session.

With that in place, hit the browser back button, and click the link:



The secret number is: **6.**

The ";jsessionid=" in the URL (in the address bar) is an indicator that a session has been created to store the value.

What happens if you hit the refresh button in the browser? Does a new number get chosen?

No … it doesn't. Remember that business about a client side redirect? Hitting the refresh button re-triggers the request URL to render the Guess page. It's not the original action request URL (the one that invokes onAction() to pick a new random number), its a render request URL to render the Guess. page.

Now that we've verified that the bucket brigade works, and that the Guess page will know about the target number, we can start filling in the rest of the behavior. The Guess page needs a property to store the most recent message (for each guess). We'll also need an index property:

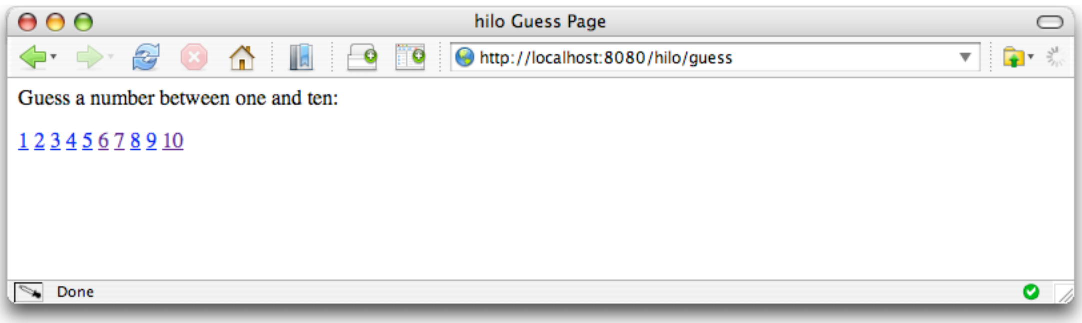
```
@Persist
```

```
private String _message;

private int _index;

public String getMessage()
{
  return _message;
}

public int getIndex()
{
  return _index;
}

public void setIndex(int index)
{
  _index = index;
}
```

These new properties are used in the Guess page's template:

```
•  src/main/webapp/WEB-INF/Guess.html
<html xmlns:t="http://tapestry.apache.org/schema/tapestry_5_0_0.xsd">
  <head>
    <title>hilo Guess Page</title>
  </head>
  <body>

    <t:if test="message">
      <p>
        <strong>${message}</strong>
      </p>
    </t:if>

    <p> Guess a number between one and ten: </p>

    <t:loop source="1..10" value="index">
      <t:actionlink context="index">${index}</t:actionlink>
    </t:loop>


  </body>
</html>
```

With this template in place, the page will render a bit differently:

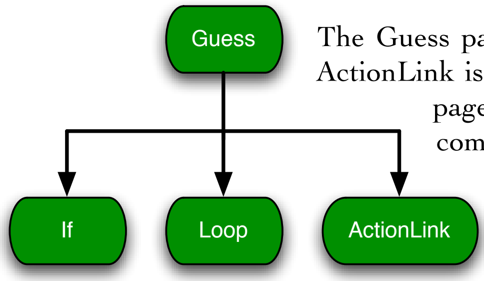


Guess a number between one and ten:

1 2 3 4 5 6 7 8 9 10

You can see that all ten options appeared as clickable links.

The template introduces two new types of components, If and Loop. It has a more complex structure than the previous examples.



The Guess page has three *embedded* components. Although the ActionLink is *enclosed* by the Loop component, from the Guess page's point of view, the If, Loop and ActionLink components are all peers of each other, as direct children of the Guess page.

The If component performs a test and, if the test is true, renders its body (the chunk of the template inside its start and end tags). The test parameter can be connected up to a boolean value, but Tapestry is flexible here. Null is considered false, and for strings (like the message property) any non-blank value is considered true. So this combines to say "if there's a message to display, then …".

The Loop component is used to iterate over a series of values. Usually, the source parameter is a property name supplying a List or array of objects, but we're using a special Tapestry shorthand for the series of numbers between 1 and 10 (inclusive).

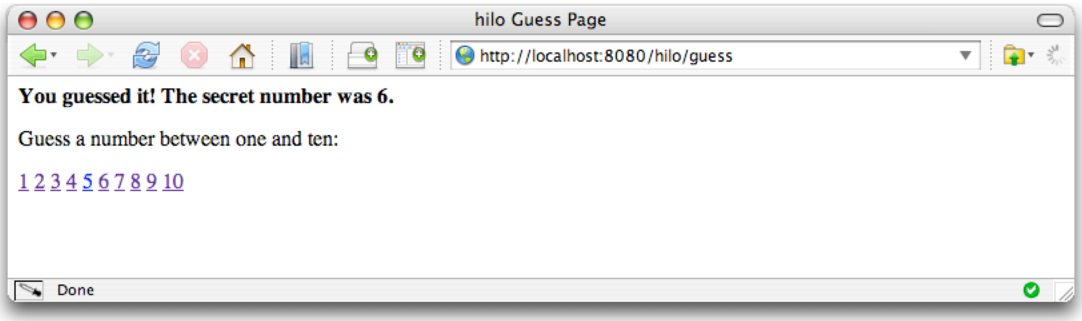
The Loop component will repeatedly render its body (the ActionLink component). It will also update its value parameter with the current value as it iterates; because the Loop component's value parameter is *bound* to the Guess page's index property, the index property will be updated.

Next is the question of how that index is communicated to the event handler method (we're about to write). That's where the context parameter of the ActionLink component comes into play: the context is a value (or a list or array of values) that are added to the action request URL. In this way, we are capturing the index value inside the link.

But how do we find out what that value is? Quite simply: our event handler method takes the value as a parameter:

```
void onAction(int guess)
{
  if (guess < _target)
    _message = String.format("%d is too low.", guess);
  else if (guess > _target)
    _message = String.format("%d is too high.", guess);
  else
    _message = String.format("You guessed it!  The secret number was ↵
%d.", guess);
}
```

Tapestry takes care of converting the context value from the URL into the matching type (int). One less thing for you to worry about. With the event handler method in place, we can actually play the game, and eventually find the secret number:



You guessed it! The secret number was 6.

Guess a number between one and ten:

1 2 3 4 5 6 7 8 9 10

Here's the final version of Guess.java:

```
•  src/main/java/org/example/hilo/pages/Guess.java
package org.example.hilo.pages;

import org.apache.tapestry.annotations.Persist;

public class Guess
{
  @Persist
  private int _target;

  @Persist
  private String _message;

  private int _index;

  public String getMessage()
  {
    return _message;
  }
  public int getIndex()
  {
    return _index;
  }

  public void setIndex(int index)
  {
    _index = index;
  }

  public int getTarget()
  {
    return _target;
  }

  void setup(int target)
  {
    _target = target;
  }

  void onAction(int guess)
  {
    if (guess < _target)
      _message = String.format("%d is too low.", guess);
    else if (guess > _target)
      _message = String.format("%d is too high.", guess);
    else
      _message = String.format("You guessed it!  The secret number was ↵
%d.", guess);
  }
}
```

That wraps it up for our hi/lo game. We could extend it in a number of ways. Here's a few challenges:

- How could we keep a running count of the number of guesses?
- Do we really need the message property, or could we compute the message on the fly? What information would we need to store in that case?
- How would we add a "try again" link? Hint: locating the code to choose the random number on the Start page may not be the best design!
- Could we change the page to only display possible remaining selections (that is, if the user chooses 8 and its too high, then we shouldn't display 8 and above anymore).

Here's a few things to reflect on before continuing:

- We've built a working web application without thinking about URLs or query parameters even once. Just our objects, our methods, and our parameters.
- The code we've written has been refreshingly simple.
- We've built a web application in Java without seeing any part of the Servlet API.
- We've been able to incrementally build our application without constantly restarting and redeploying.
- When we've made minor mistakes, Tapestry has provided us with very detailed information needed to correct the error.

Next up: Forms, input validation, and a little bit more magic!

Chapter 4

Building forms with Tapestry

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.