Tapestry Tutorial

by Howard Lewis Ship

Tapestry Tutorial

by Howard Lewis Ship

Table of Contents

List of Figures

Chapter 1. Introduction

Warning

This Tutorial is extremely out of date. A new tutorial should be ready before 3.0 reaches GA.

Tapestry is a new application framework for developing web applications. It uses a component object model to represent the pages of a web application. This is similar to spirit to using the Java Swing component object model to build GUIs.

Just like using a GUI toolkit, there's some preparation and some basic ideas that must be cleared before going to more ambitious things. Nobody writes a word processor off the top of their head as their first GUI project; nobody should attempt a full-featured e-commerce site as their first attempt using Tapestry.

The goal of Tapestry is to eliminate most of the coding in a web application. Under Tapestry, nearly all code is directly related to application functionality, with very little "plumbing". If you have previously developed a web application using Microsoft Active Server Pages, JavaServer Pages or Java Servlets, you may take for granted all the plumbing: writing servlets, assembling URLs, parsing URLs, managing objects inside the `HttpSession`, etc.

Tapestry takes care of nearly all of that, for free. It allows for the development of rich, highly interactive applications.

This tutorial will start with basic concepts, such as the "Hello World" application, and will gradually build up to more sophisticated examples.

The tutorial uses Jetty, a freely available servlet engine, which is packaged with the Tapestry distribution.

The format of this tutorial is to describe (visually and with text) an application within the tutorial, then describe how it is constructed, using code excerpts. The reader is best served by having an IDE open so that they can look at the code in detail, as well as run the applications.

Chapter 2. Setting up the Tutorial

This document expects that you will have extracted the full Tapestry distribution to your `C:` drive [1]

This will have created a directory `C:\Tapestry-x.x` and, beneath it, several more directories. [2]

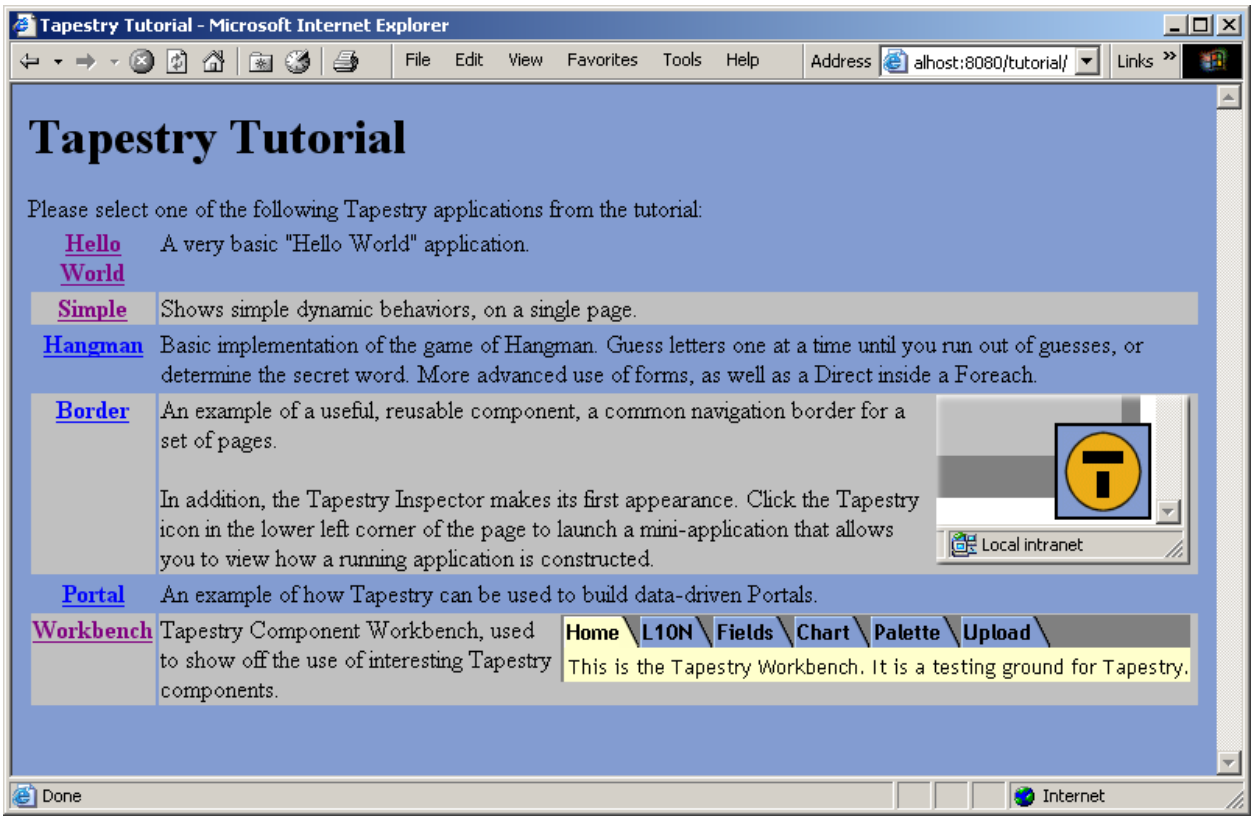
The source code for the Tutorial is distributed as a JAR file, `src/examples-src.jar`. A precompiled WAR file, `lib/tutorial.war` is included in the distribution.

The Tapestry distribution includes an Ant build file that allows the Tutorial to be directly executed. Ant release 1.5 is required.

From the Tapestry root directory, execute the command **ant -emacs run-tutorial**, which will launch the Jetty server for the Tutorial.

Once Jetty is running, you can access the Tutorials using the URL `http://localhost:8080/tutorial`.

Figure 2.1. Tutorial Index Page



1 If you are using Solaris or another non-Windows operating system, you're expected to be savvy enough to translate to a sensibly constructed file system.
2 The three numbers are the release number. At the time of this writing, the release was 2.2, but this is constantly changing. Simply adjust the actual pathname to reflect the release of Tapestry you downloaded.

Chapter 3. Hello World

In this first example, we'll create a very simple "Hello World" kind of application. It won't have any real functionality but it'll demonstrate the simplest possible variation of a number of key aspects of the framework.

We'll define our application, define the lone page of our application, configure everything and launch it.

The code for this section of the tutorial is in the Java package `tutorial.hello`, i.e., `C:\Tapestry-x.x.x\examples\Tutorial\src\tutorial\hello`.

Application Engine

As each new client connects to the application, an instance of the application engine is created for them. The application engine is used to track that client's activity within the application.

The application engine is an instance, or subclass of, the Tapestry class `SimpleEngine`.

In these first few examples, we have no additional behavior to add to the provided base class, so we simply use `SimpleEngine` directly.

Web Deployment Descriptor

The application servlet is a "bridge" between the servlet container and the application engine. Its job is simply to create (on the first request) or locate (on subsequent requests) the application engine.

All Tapestry applications use the same servlet class, however its configuration is different. Part of the configuration is to identify the location of the *application specification* which is like a master index of all the pages in the application.

The tutorial is a rare case; it is a single WAR that contains multiple Tapestry applications. This isn't a problem ... each Tapestry application has its own servlet and has its own configuration. The following figure shows the deployment descriptor for the Tapestry Tutorial (but excludes the additional sections for the other applications within the WAR).

Figure 3.1. Tutorial Deployment Descriptor (partial)

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
  "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
  <display-name>Tapestry Tutorial</display-name>

  <servlet>
    <servlet-name>hello</servlet-name>
    <servlet-class>org.apache.tapestry.ApplicationServlet</servlet-class>
    <init-param>
            <param-name>org.apache.tapestry.application-specification</param-name>
            <param-value>/tutorial/hello/HelloWorld.application</param-value>
    </init-param>
          <load-on-startup>0</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>hello</servlet-name>
    <url-pattern>/hello</url-pattern>
  </servlet-mapping>
```

```
  <session-config>
          <session-timeout>15</session-timeout>
  </session-config>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>
</web-app>
```

Application Specification

The application specification is used to describe the application to the Tapestry framework. It provides the application with a name, an engine class, and a list of pages.

This specification is a file that is located on the Java class path. In a deployed Tapestry application, the specification lives with the application's class files, in the `WEB-INF/classes` directory of a War file.

Figure 3.2. HelloWorld.application

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
  "-//Howard Lewis Ship//Tapestry Specification 1.3//EN"
  "http://tapestry.sf.net/dtd/Tapestry_1_3.dtd">
<application
  name="Hello World Tutorial"
  engine-class="org.apache.tapestry.engine.SimpleEngine">

  <page name="Home"
    specification-path="/tutorial/hello/Home.page"/>

</application>
```

Our application is very simple; we give the application a name, use the standard engine, and define a single page, named "Home". In Tapestry, pages and components are specified with the path to their specification file (a file that end with '.page' for page specifications or '.jwc' for component specifications).

Page "Home" has a special meaning to Tapestry: when you first launch a Tapestry application, it loads and displays the "Home" page. All Tapestry applications are required to have such a home page.

Home Page Specification

The page specification defines the Tapestry component responsible for the page. In this first example, our component is very simple.

Figure 3.3. Home.page

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification PUBLIC
  "-//Howard Lewis Ship//Tapestry Specification 1.3//EN"
  "http://tapestry.sf.net/dtd/Tapestry_1_3.dtd">

<page-specification class="org.apache.tapestry.html.BasePage"/>
```

This simply says that `Home` is a kind of page. We use the supplied Tapestry class `BasePage` since we aren't adding any behavior to the page.

Home Page Template

Finally, we get to the content of our application. This file is also a Java resource; it isn't directly visible to the web server. It has the same location and name as the component specification, except that it ends in "html".

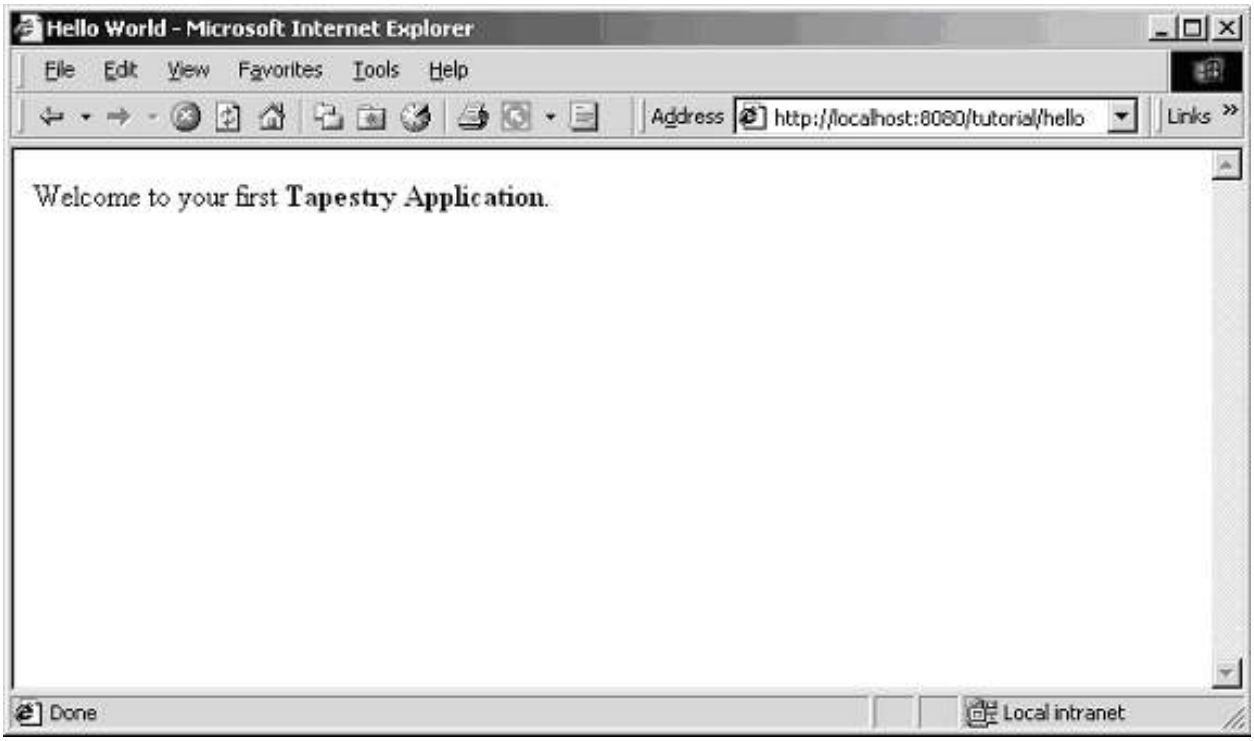
Figure 3.4. Home.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
<head>
        <title>Hello World</title>
</head>

<body>

Welcome to your first <b>Tapestry Application</b>.

</body>
</html>
```

Run the Application

You should already be running the Jetty server in a window, and have a browser running the tutorials page. Select the first option, Hello World, from the list. You will be presented with the first (and only) page generated by Tapestry for this application:

Figure 3.5. Hello World Application



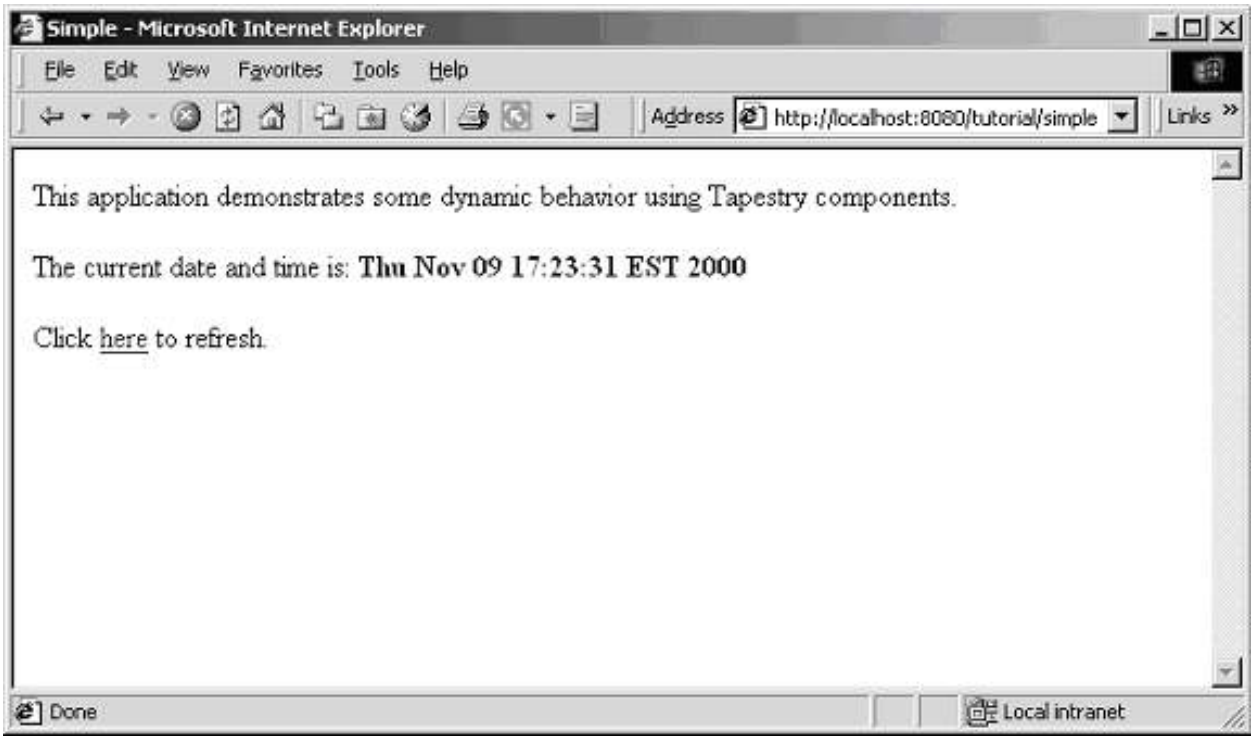
Not much of an application ... there's no interactivity. It might as well be a static web page, but it's a start. Remember, there was no JavaServer page here, and no HTML directly visible to the web server. We used the Tapestry framework to assembly an application consisting of a single component.

In the following chapters, we'll see how to add dynamic content and then true interactivity.

Chapter 4. Dynamic Content

In this chapter, we'll create a new web application that will show some dynamic content. We'll also begin to show some interactivity by adding a link to the page. Our dynamic content will simply be to show the current date and time. The interactivity will be a link to refresh the page. It all looks like this:

Figure 4.1. Dynamic Application



Clicking the word "here" will update the page showing the new data and time. Not incredibly interactive, but it's a start.

The code for this section of the tutorial is in the package `tutorial.simple`.

The application specification is almost identical to the Hello World example:

Figure 4.2. Simple.application

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
  "-//Howard Lewis Ship//Tapestry Specification 1.3//EN"
  "http://tapestry.sf.net/dtd/Tapestry_1_3.dtd">

<application name="Simple Tutorial" engine-class="org.apache.tapestry.engine.SimpleEngine">
  <page name="Home" specification-path="/tutorial/simple/Home.page"/>
</application>
```

Things only begin to get more interesting when we look at the HTML template for the home page:

Figure 4.3. Home.html

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
        <title>Simple</title>
</head>
<body>

This application demonstrates some dynamic behavior using Tapestry components.

<p>The current date and time is: <b><span jwcid="insertDate">Current Date</span></b>

<p>Click <a jwcid="refresh">here</a> to refresh.

</body>
</html>
```

This looks like ordinary HTML, except for the special `jwcid` attribute. "jwc" is short for "Java Web Component"; these attributes identify the tag as a placeholder for a dynamic Tapestry component.

We have two components. The first inserts the current date and time into the HTML response. The second component creates a hyperlink that refreshes the page when clicked.

One of the goals of Tapestry is that the HTML should have the minimum amount of special markup. This is demonstrated here ... the dynamic component tags blend into the standard HTML of the template. We also don't confuse the HTML by explaining exactly what an `insertDate` or `refresh` is; that comes out of the specification (described shortly). The ids used here are meaningful only to the developer [3], the particular type and configuration of each component is defined in the component specification.

Tapestry doesn't really care what HTML tag you use, as long as you balance the tag correctly. In fact, it ignores the tag entirely: the `refresh` component above could just has easily been identified with a tag, or any other tag for that matter. Tapestry is only interested in the *structure* of the HTML template. The fact that you can use meaningful tags is a convienience; it allows a Tapestry HTML template to be previewed in a WYSIWYG HTML editor, such as HomeSite. Additionally, Tapestry edits out the content of tags for components that don't wrap around other content: the `insertDate` component in this example. This allows a preview values to be kept in the template.

Very significant is the fact that a Tapestry component can *wrap* around other elements of the template. The `refresh` component wraps around the word "here". What this means is that the `refresh` component will get a chance to emit some HTML (an `<a>` hyperlink tag), then emit the HTML it wraps (the word "here"), then get a chance to emit more HTML (the `</a>` closing tag).

What's more important is that the component can not only wrap static HTML from the template (as shown in this example), but may wrap around other Tapestry components and those components may themselves wrap text and components, to whatever depth is required.

And, as we'll see in later chapters, a Tapestry component itself may have a template and more components inside of it. In a real application, the single page of HTML produced by the framework may be the product of dozens of components, effectively "woven" from dozens of HTML templates.

Again, the HTML template doesn't define what the components are, it is simply a mix of static HTML that will be passed directly back to the client web browser, with a few placeholders (the tags with the `jwcid` attribute) for where dynamic content will be plugged in.

The page's component specification defines what types of components are used and how data moves between the application, page and any components.

[3] Of course, good and consistent naming is important.

Figure 4.4. Home.page

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification PUBLIC
  "-//Howard Lewis Ship//Tapestry Specification 1.3//EN"
  "http://tapestry.sf.net/dtd/Tapestry_1_3.dtd">

<page-specification class="tutorial.simple.Home">

  <component id="insertDate" type="Insert">
    <binding name="value" expression="currentDate"/>
  </component>

  <component id="refresh" type="PageLink">
    <static-binding name="page">Home</static-binding>
  </component>

</page-specification>
```

Here's what all that means: The `Home` page is implemented with a custom class, `tutorial.simple.Home`. It contains two components, `insertDate` and `refresh`.

The two components used within this page are provided by the Tapestry framework.

The `insertDate` component is type `Insert`. `Insert` components have a `value` parameter used to specify what should be inserted into the HTML produced by the page. The `insertDate` component has its `value` parameter bound to a JavaBeans property of its container (the page), the `currentDate` property.

The `refresh` component is type `PageLink`, meaning it creates a link to some other page in the application. `PageLink` components have a parameter, named `page`, which defines the name of the page to navigate to. The name is matched against a page named in the application specification.

In this case, we only have one page in our application (named "Home"), so we can use a static binding for the page parameter. A static binding provides a value for the component parameter statically, the same value every time. The value is defined right in the specification.

That just leaves the implementation of the Home page component:

Figure 4.5. Home.java

```
package tutorial.simple;

import java.util.Date;
import org.apache.tapestry.html.BasePage;

public class Home extends BasePage
{
  public Date getCurrentDate()
  {
    return new Date();
  }
}
```

`Home` implements a read-only JavaBeans property, `currentDate`. This is the same `currentDate` that the `insertDate` component needs. When asked for the current date, the `Home` object returns a new instance of the `java.util.Date` object.

The `insertDate` component converts objects into strings by invoking `toString()` on the object.

Now all the bits and pieces are working together.

Run the application, and use the View Source command to examine the HTML generated by by framework.

Figure 4.6. HTML generated for Home page

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
        <title>Simple</title>
</head>
<body>

This application demonstrates some dynamic behavior using Tapestry components.

<p>The current date and time is: <b>Fri Nov 23 17:05:53 PST 2001</b>

<p>Click <a href="/tutorial/simple...">here</a> to refresh.

</body>
</html>
```

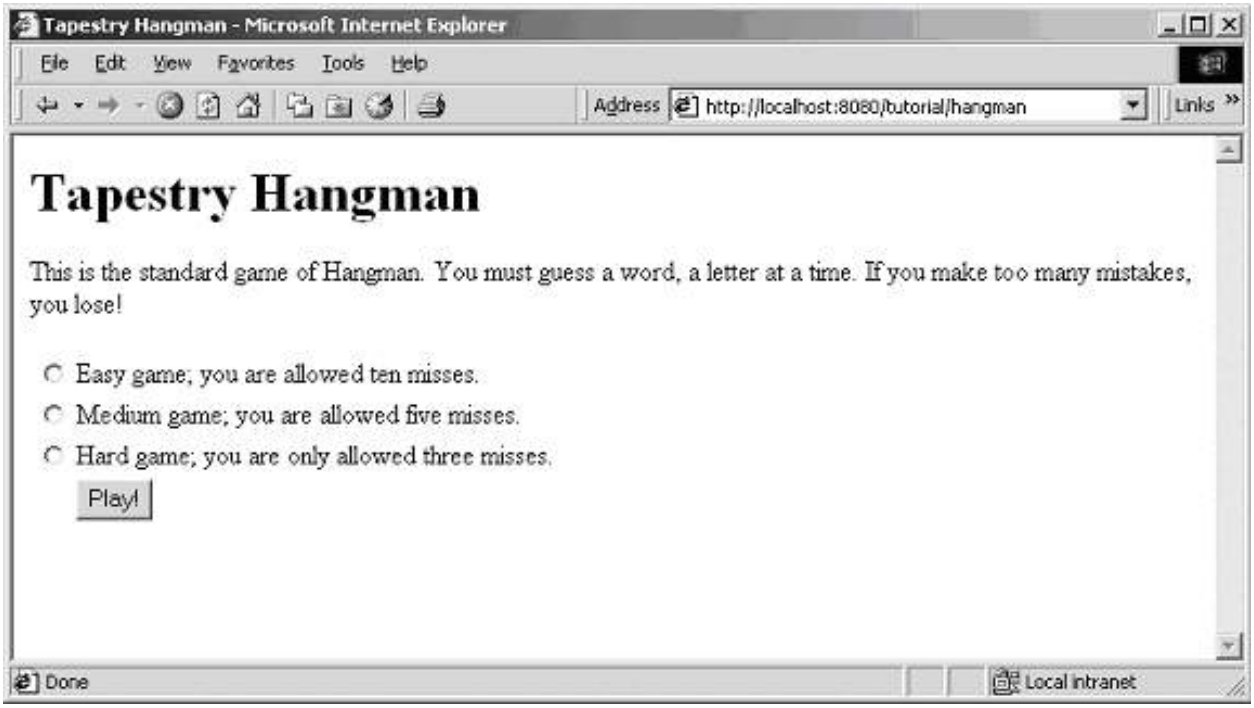
This should look very familiar, in that it is mostly the same as the HTML template for the page. Tapestry not only inserted simple text (the current date and time, obtained from an `java.util.Date` object), but the `refresh` component inserted the `<a>` and `</a>` tags, and created an appropriate URL for the href attribute.

Chapter 5. Hangman

So far, these examples have been a little bit cut-and-dried. Lets do a meatier example that uses a few more interesting components. Let's play Hangman!

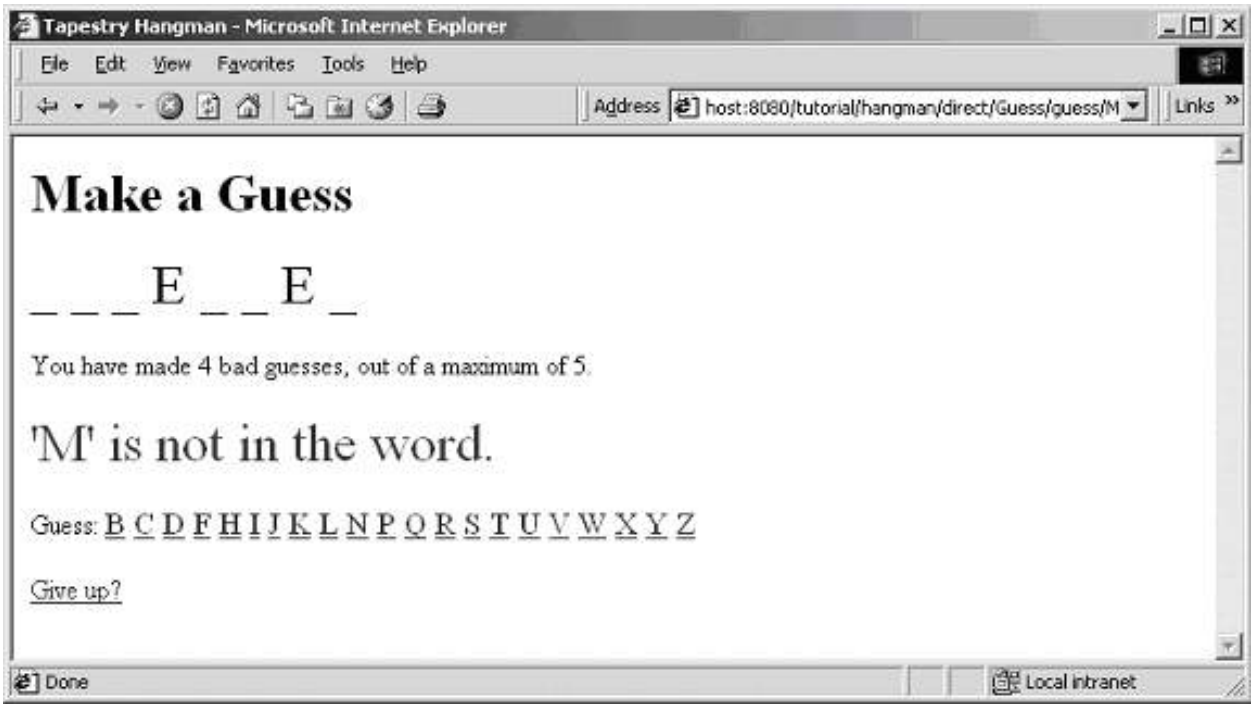
Our Hangman application consists of four pages. The Home page allows a new game to be started, which includes selecting the difficulty of the game (how many wrong guesses you are allowed).

Figure 5.1. Hangman Home Page



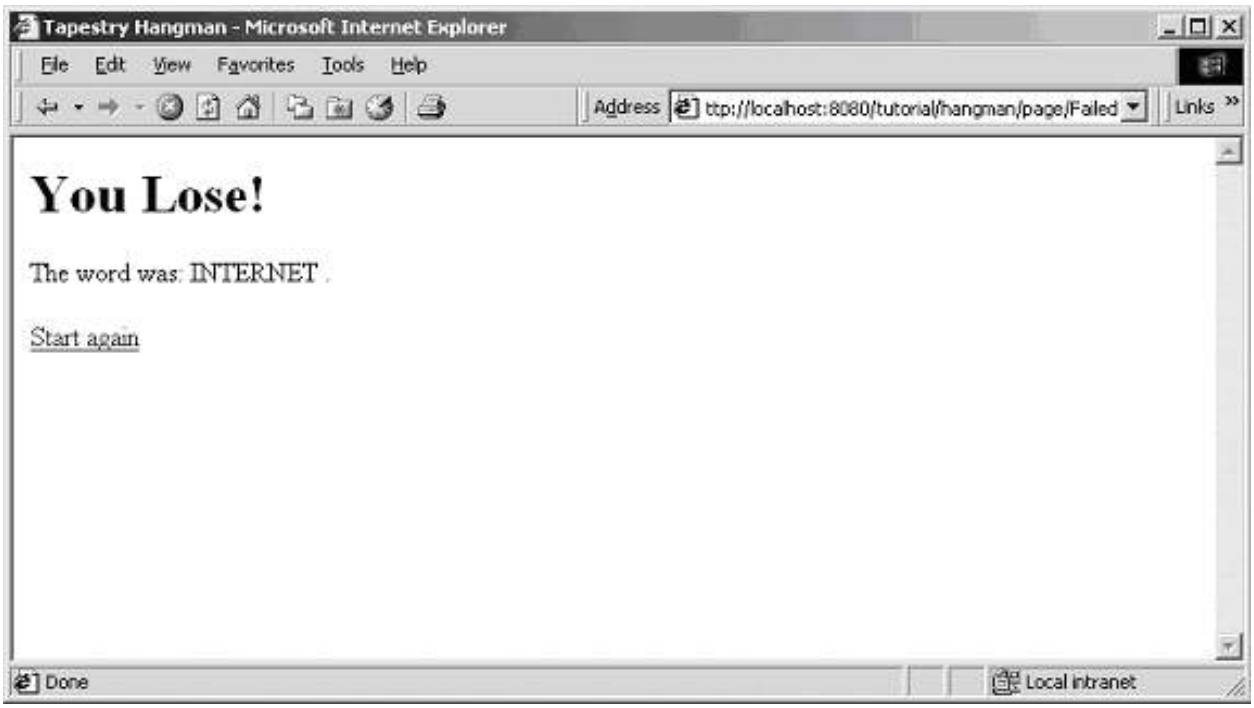
The main page is the Guess page, where the partially filled out word is displayed, and the user can make guesses (from a shrinking list of possible letters):

Figure 5.2. Hangman Guess Page



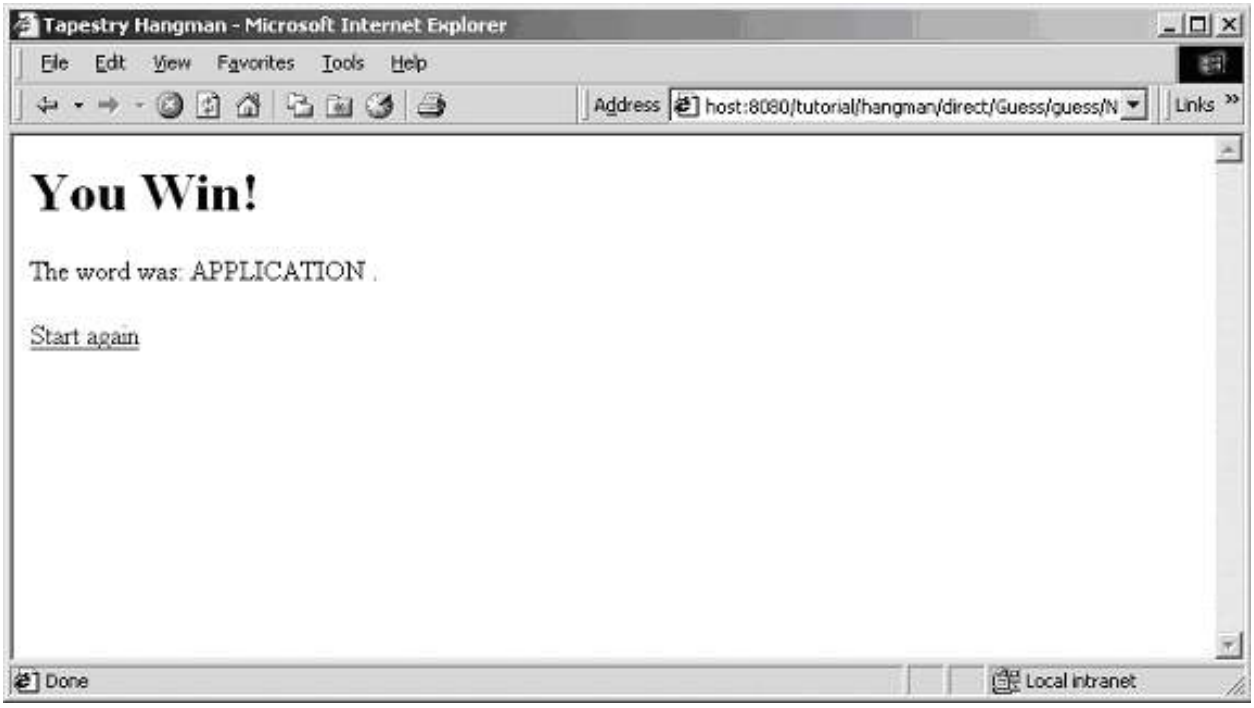
After you give up, or when you make too many mistakes, you end up on the the Failed page:

Figure 5.3. Hangman Failed Page



But, if you guess all the letters, you are sent to the Success page:

Figure 5.4. Hangman Success Page



The Visit Object

The center of this application is an object that represents game, an object of class `HangmanGame`. This object is used to track the word being guessed, the letters that have been used, the number of misses and the letters that have been correctly guessed.

This object is a property of the *visit* object. What's the visit object? The visit object is a holder of all information about a single client's visit to your web application. It contains data and methods that are needed by the pages and components of your application.

The visit object is owned and created by the engine object. It is serialized and de-serialized with the engine.

The application specification includes a little extra segment at the bottom to specify the class of the visit object.

Figure 5.5. Hangman.application

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
  "-//Howard Lewis Ship//Tapestry Specification 1.3//EN"
  "http://tapestry.sf.net/dtd/Tapestry_1_3.dtd">

<application name="Tapestry Hangman" engine-class="org.apache.tapestry.engine.SimpleEngine">

  <property name="org.apache.tapestry.visit-class">tutorial.hangman.Visit</property> ❶

  <page name="Home" specification-path="/tutorial/hangman/Home.page"/>

  <page name="Guess" specification-path="/tutorial/hangman/Guess.page"/>
```

```
  <page name="Failed" specification-path="/tutorial/hangman/Failed.page"/>
  <page name="Success" specification-path="/tutorial/hangman/Success.page"/>
</application>
```

❶ This property specifies that the engine should instantiate an instance of `tutorial.hangman.Visit` when a visit object is first required. This is the default way in which the visit object is specified, though if the visit object doesn't have an empty constructor method, the engine method `createVisit()` must be implemented instead.

So, returning from that distraction, the game object is a property of the visit object, which is accessible from any page (via the page's visit property).

The Home Page

The Home page's job is to collect the difficulty and initiate a game:

Figure 5.6. Home.java

```
public class Home extends BasePage
{
        public static final int EASY = 10;
        public static final int MEDIUM = 5;
        public static final int HARD = 3;

        private int misses;
        private String error;

        public void detach()
        {
                misses = 0;
                error = null;

                super.detach();
        }

        public int getMisses()
        {
                return misses;
        }

        public void setMisses(int value)
        {
                misses = value;

                fireObservedChange("misses", value);
        }

        public String getError()
        {
                return error;
        }

        public void formSubmit(IRequestCycle cycle)
        {
                if (misses == 0)
                {
                        error = "Please select a game difficulty.";
                        return;
                }
```

```
                Visit visit = (Visit) getVisit();

                visit.start(misses);

                cycle.setPage("Guess");
        }
}
```

We're seeing all the familiar ideas: The `misses` property is a persistent page property (which means the page will "remember" the value previously selected by the user).

We use a common trick for simple pages: the page contains a single `Form` component, so we use the page itself as the form's listener, and have the page implement the `IActionListener` interface.

This saves a bit of code for creating an inner class as the form listener.

Initially, we don't select a difficulty level, and the user can click "Play!" without selecting a value from the list, so we check that.

Otherwise, we get the visit object and ask it to start a new game with the selected number of misses. We then jump to the Guess page to start accepting guesses from the user.

The interesting part of the Home page HTML template is the form:

Figure 5.7. Home.html (excerpt)

```
<form jwcid="form">

<span jwcid="group">

<span jwcid="ifError">
<font size=+2 color=red><span jwcid="insertError"/></font>
</span>

<table>
        <tr>
                <td><input jwcid="inputEasy"/></td>
                <td>Easy game; you are allowed ten misses.</td>
        </tr>

        <tr>
                <td><input jwcid="inputMedium"/></td>
                <td>Medium game; you are allowed five misses.</td>
        </tr>

        <tr>
                <td><input jwcid="inputHard"/></td>
                <td>Hard game; you are only allowed three misses.</td>
        </tr>

        <tr>
                <td></td>
                <td><input type="submit" value="Play!"></td>
        </tr>

</table>

</span>
</form>
```

Here, the interesting components are `group`, `inputEasy`, `inputMedium` and `inputHard`. `group` is type `RadioGroup`, a wrapper that must go around the `Radio` components (the other three). The `RadioGroup` determines what property of the page is to be read and updated (its bound to the `misses` property). Each `Radio` button is associated with a particular value to be assigned to the property, when that radio button is selected by the user.

This comes together in the Home page specification:

Figure 5.8. Home.page (excerpt)

```
<component id="group" type="RadioGroup">
  <binding name="selected" expression="misses"/>
</component>

<component id="inputEasy" type="Radio">
  <field-binding name="value" field-name="tutorial.hangman.Home.EASY"/> ❶
</component>

<component id="inputMedium" type="Radio">
  <field-binding name="value" field-name="tutorial.hangman.Home.MEDIUM"/>
</component>

<component id="inputHard" type="Radio">
  <field-binding name="value" field-name="tutorial.hangman.Home.HARD"/>
</component>
```

❶ A `<field-binding>` is like a `<static-binding>`, except that the static value is taken from a public static field of some class. This makes it easy to coordinate behaviors between the specification and the class.

This is a good thing, since if you decide to make a `HARD` game only allow two mistakes, you can make the change in exactly one place .. your Java code.

So the end result is: when the user clicks the radio button for a Hard game, the static constant `HARD` is assigned to the page's `misses` property.

The Guess Page

This is the page where uses make letter guesses. The page has four sections:

- A display of the word, with underscores replacing unguessed letters.
- A status area, showing the number of bad guesses and an optional error message after an invalid guess.
- A list of letters that may be guessed. Letters disappear after they are used.
- An option to give up and see the word, terminating the game.

Let's start with the HTML template this time:

Figure 5.9. Guess.html (excerpt)

```
<h1>Make a Guess</h1>
```

```
<font size=+3>
        <span jwcid="insertGuess"/>
</font>

<p>

You have made <span jwcid="insertMissed"/> bad guesses,
out of a maximum of <span jwcid="insertMaxMisses"/>.

<span jwcid="ifError">
<p>
<font size=+3 color=red><span jwcid="insertError"/></font>
</span>

<p>Guess:
<font size=+1>
<span jwcid="e">
<a jwcid="guess"><span jwcid="insertLetter"/></a>
</span>
</font>

<p><a jwcid="giveUp">Give up?</a>
```

Most of these components should be fairly obvious by now; let's focus on the components that allow the user to guess a letter. This could have been implemented in a number of ways using more radio buttons, a drop down list or a text field the user could type into. In this example, we chose to simply create a series of links, one for each letter the user may still guess.

Let's look at the specification for those three components (`e`, `guess` and `insertLetter`).

Figure 5.10. Guess.jwc (excerpt)

```
<component id="e" type="Foreach">
  <binding name="source" expression="unused"/>
</component>

<component id="guess" type="DirectLink">
  <binding name="listener" expression="listeners.makeGuess"/>
  <binding name="parameters" expression="components.e.value"/>
</component>

<component id="insertLetter" type="Insert">
  <binding name="value" expression="components.e.value"/>
</component>
```

Component `e` is simply a `Foreach`, the `source` is the `unused` property of the page (we'll see in a moment how the page gets this list of unused letters from the game object).

Component `insertLetter` inserts the current letter from the list of unused letters. It gets this current letter directly from the `e` component. On successive iterations, a `Foreach` component's `value` property is the value for the iteration.

Component `guess` is type `DirectLink`, which creates a hyperlink on the page and notifies its listener when the user clicks the link. Just knowing that the component was clicked isn't very helpful though; the application needs to know which letter was actually clicked.

Passing that kind of information along is accomplished by setting the `parameters` parameter for the component. The `parameters` parameter is an object, or array or objects, that will be encoded into the URL for the hyperlink. When the component's listener is notified, it can obtain the array of objects from the `IRequestCycle` [4].

4 Tapestry takes care of converting objects into strings when constructing the URL, then converts those strings back into objects when the link is clicked. Your listener method will be able to get *copies* of the original parameters.

These *service parameters* are often used to encode primary keys of objects, names of columns or other information specific to the application.

In this case, the service parameters consist of a single value, the letter to be guessed.

All of this comes together in the Java code for the Guess page.

Figure 5.11. Guess.java (excerpt)

```
public void makeGuess(IRequestCycle cycle)
{
    Object[] parameters = cycle.getServiceParameters();
        char letter = ((Character)parameters[0]).charValue();
        HangmanGame game = getGame();

        try
        {
                game.guess(letter);
        }
        catch (GameException ex)
        {
                error = ex.getMessage();

                if (game.getFailed())
                        cycle.setPage("Failed");

                return;
        }

        // A good guess.

        if (game.getDone())
                cycle.setPage("Success");
}
```

The component specification showed how data was encoded into the URL as the service parameters; here we see how the `makeGuess()` listener method has access to the service parameters and uses them. The listener method extracts the letter and informs the game object, which throws an exception if the letter is not in the word being guessed.

The method `HangmanGame.getFailed()` returns `true` when all the missed guesses are used up, at which point we go to the `Failed` page to tell the user what the word was.

On the other hand, if an exception isn't thrown, then the guess was good. `getDone()` returns `true` if all letters have been guessed, in which go to the `Success` page.

If all letters weren't guessed, we stay on the `Guess` page, which will display the word with the guessed letter filled in, and with fewer options in the list of possible guesses.

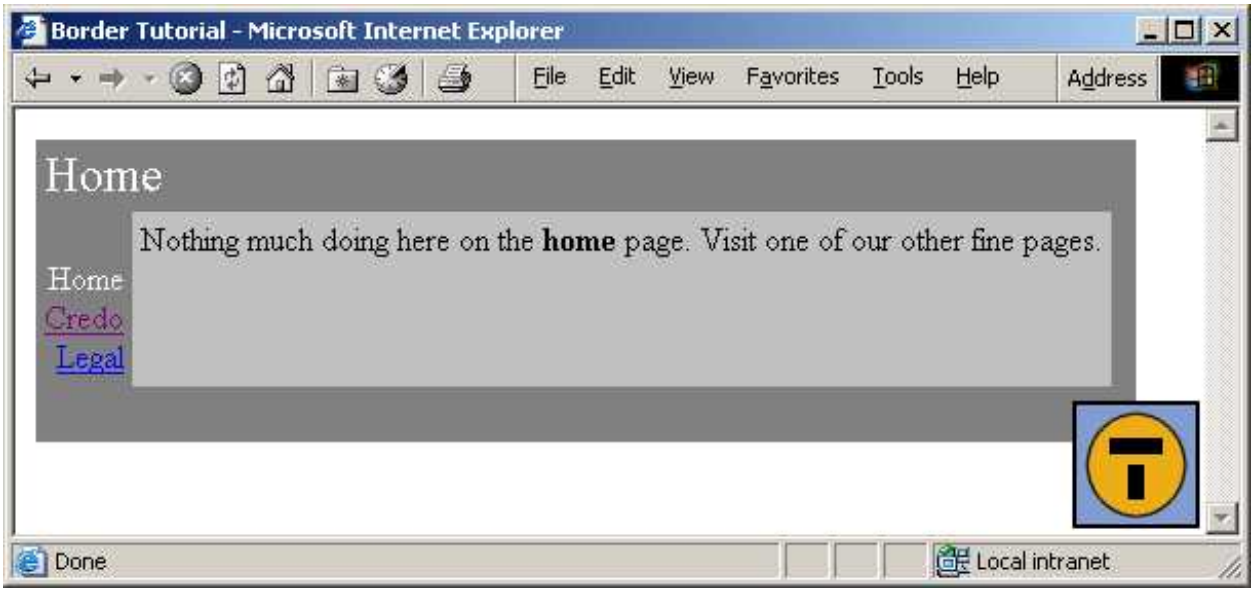
Limitations

This is a very, very simple implementation of the game. For example, it's easy to cheat; you can give up, then use your browser's back button to return to the `Guess` page and keep guessing (with accuracy, if your memory is any good).

Chapter 6. Creating Reusable Components

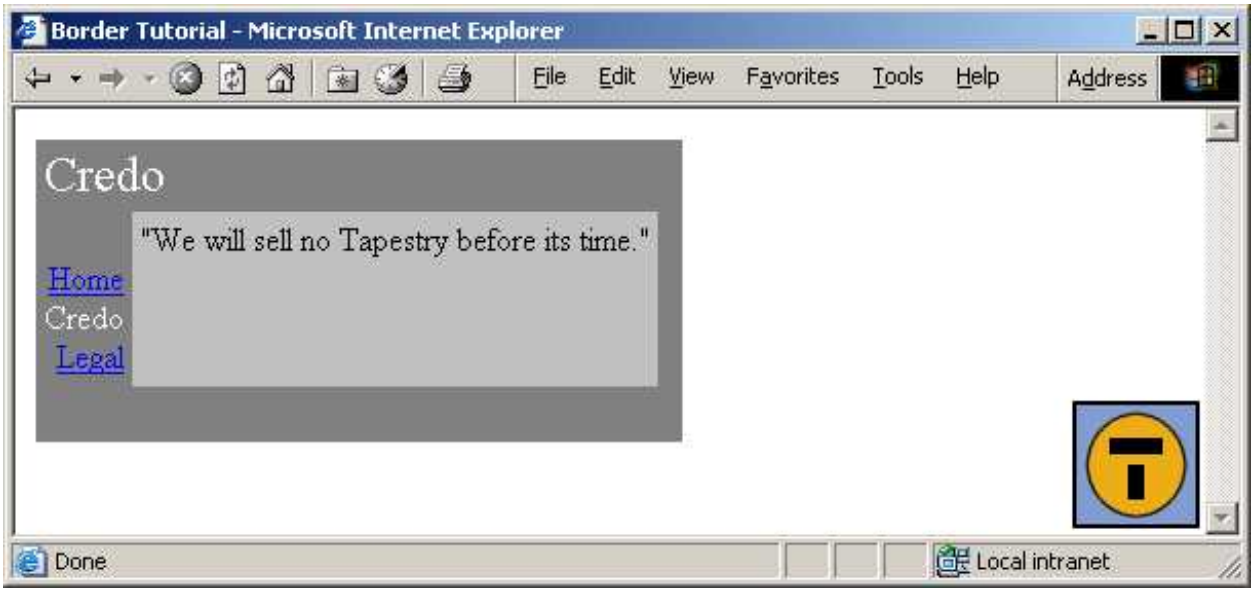
In this tutorial, we'll show how to create a reusable component. One common use of components it to create a common "border" for the application that includes basic navigation. We'll be creating a simple, three page application with a navigation bar down the left side.

Figure 6.1. Border Home Page



Navigating to another page results in a similar display:

Figure 6.2. Border Credo Page



Each page's content is confined to the silver area in the center. Note that the border adapts itself to each page: the title "Home" or "Credo" is specific to the page, and the current page doesn't have an active link (in the above page, "Credo" is the current page, so only "Home" and "Legal" are usable as navigation links).

The "i" in the gear is the Show Inspector link. It will be described in the next chapter.

Because this tutorial is somewhat large, we'll only be showing excerpts from some of the files. The complete source of the tutorial examples is available seperately, in the `tutorial.border` package.

Each of the three pages has a similar HTML template:

Figure 6.3. Home.html

```
<span jwcid="border">

Nothing much doing here on the <b>home</b> page.  Visit one of our other
fine
pages.

</span>
```

Remember that Tapestry components can wrap around other HTML elements or components. For the border, we have an HTML template where everything on the page is wrapped by the `border` component.

Note that we don't specify any `<html>` or `<body>` tags; those are provided by the `Border` component (as well as the matching close tags).

This illustrates a key concept within Tapestry: embedding vs. wrapping. The `Home` page embeds the `border` component (as we'll see in the `Home` page's specification). This means that the `Home` page is implemented using the `border` component.

However, the `border` component wraps the content of the `Home` page, the `Home` page HTML template indicates the *order* in which components (and static HTML elements) are renderred. On the `Home` page, the `border` component 'bats' first and cleanup.

The construction of the `Border` component is driven by how it differs from page to page. You'll see that on each page, the title (in the upper left corner) changes. The names of all three pages are displayed, but only two of the three will have links (the third, the current page, is just text). Lastly, each page contains the specific content from its own HTML template.

Figure 6.4. Border.html

```
<span jwcid="shell"> ❶
<body jwcid="body"> ❷
<table border=0 bgcolor=gray cellspacing=0 cellpadding=4>
  <tr valign=top>
    <td colspan=3 align=left>
      <font size=5 color="White"><jwc id="insertPageTitle"/></font>
    </td>
  </tr>
  <tr valign=top>
    <td align=right>
      <font color=white>
<span jwcid="e"> ❸
        <br><a jwcid="link"><span jwcid="insertName"/></a> ❹
</span>
```

```
      </font>
    </td>
    <td rowspan=2 valign=top bgcolor=silver>
      <span jwcid="renderBody"/> ❺
    </td>
    <td rowspan=2 width=4></td>
  </tr>
  <tr>
       <td><span jwcid="inspector"/></td> ❻
  </tr>
  <tr>
    <td colspan=3 height=4> </td>
  </tr>
</table>
</body>
</span>
```

❶ The `shell` component provides the `<html>` and `<head>` elements of the response HTML.
❷ The `body` components provides the `<body>` element. It also provides support for JavaScript related to `Rollover` buttons, such as the `showInspector` component.
❸ The `e` component is a `Foreach` configured to work through a list of page names (provided by the engine).
❹ The `link` and `insertName` components provide the inter-page navigation links.
❺ The `renderBody` component provides the actual content for the page. The `Border` component is used on all three pages, but its a different instance on each page, wrapping around different content specific to the page.
❻ The `showInspector` component provides the button below the page names (the italicized "i" in a circle) and will be explained shortly.

The `Border` component is designed to be usable in other Tapestry applications, so it doesn't hard code the list of page names. These must be provided to the component as a parameter. In fact, the application engine provides the list.

Figure 6.5. Border.jwc

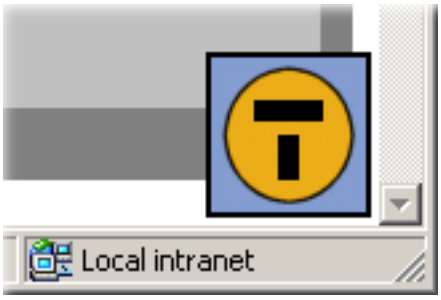
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE component-specification PUBLIC
  "-//Howard Lewis Ship//Tapestry Specification 1.3//EN"
  "http://tapestry.sf.net/dtd/Tapestry_1_3.dtd">

<component-specification class="tutorial.border.Border" allow-informal-parameters="no">

  <parameter name="title" java-type="java.lang.String" required="yes"/> ❶

  <parameter name="pages" required="yes"/> ❷

  <component id="shell" type="Shell">
    <binding name="title" expression="page.engine.specification.name"/> ❸
  </component>

  <component id="insertPageTitle" type="Insert">
    <inherited-binding name="value" parameter-name="title"/> ❹
  </component>

  <component id="body" type="Body"/>

  <component id="e" type="Foreach"> ❺
    <inherited-binding name="source" parameter-name="pages"/>
    <binding name="value" expression="pageName"/>
  </component>

  <component id="link" type="PageLink"> ❻
    <binding name="page" expression="pageName"/>
    <binding name="disabled" expression="disablePageLink"/>
```

```
  </component>

  <component id="insertName" type="Insert">
    <binding name="value" expression="pageName"/>
  </component>

  <component id="renderBody" type="RenderBody"/>

  <component id="inspector" type="InspectorButton"/> ❼
</component-specification>
```

❶ Declares a required parameter for the border, the title that will appear on the page.
❷ Declares a parameter to specify the list of page names. We don't specify a particular type because its pretty unbounded; the framework will accept `List`, `Iterator` or a Java array.
❸ We then provide the `shell` component with its `title` parameter; this will be the window title. We use the application's name, with is extracted from the application's specification.
❹ The `<inherited-binding>` element allows a component to share its parameters. Here the `Border`'s `title` is used as the `value` parameter of the `insertPageTitle` component (an `Insert`). Using these inherited bindings simplifies the process of creating complex components from simple ones.
❺ Likewise, the `e` component (a `Foreach`) needs as its source the list of pages, which it inherits from the `Border` component's `pages` parameter. It has been configured to store each succesive page name into the `pageName` property of the `Border` component; this is necessary so that the `Border` component can determine which page link to disable (it disables the current page since we're already there).
❻ The `link` component creates the link to the other pages. It has a `disabled` parameter; which, when true, causes the link component to not create the hyperlink (though it still allows the elements it wraps to render). The Java class for the `Border` component, `tutorial.border.Border`, provides a method, `getDisablePageLink()`, that returns true when the `pageName` instance variable (set by the `e` component) matches the current page's name.
❼ This component will raise the Tapestry Inspector in a new window when clicked.

So, the specification for the `Border` component must identify the parameters it needs, but also the components it uses and how they are configured.

Figure 6.6. Show Inspector Button



Clicking on the button raises a second window that describes the current page in the application (this is used when debugging a Tapestry applicaton). The Inspector is described in the next chapter.

The final mystery is the `wrapped` component. It is used to render the elements wrapped by the `Border` on the page containing the `Border`. Those elements will vary from page to page; running the application shows that they are different on the home, credo and legal pages (different text appears in the central light-grey box). There is no limitation on the elements either: Tapestry is specifically designed to allow components to wrap other components in this way, without any arbitrary limitations.

This means that the different pages could contain forms, images or any set of components at all, not just static

HTML text.

The specification for the home page shows how the title and pages parameters are set. The title is static, the literal value "Home" (this isn't the best approach if localization is a concern).

Figure 6.7. Home page specification

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification PUBLIC
  "-//Howard Lewis Ship//Tapestry Specification 1.3//EN"
  "http://tapestry.sf.net/dtd/Tapestry_1_3.dtd">

<page-specification class="org.apache.tapestry.html.BasePage">

  <component id="border" type="Border">
    <static-binding name="title">Home</static-binding>
    <binding name="pages" expression="engine.pageNames"/>
  </component>

</page-specification>
```

The `pages` property is retrieved from the application engine, which implements a `pageNames` JavaBeans property:

Figure 6.8. BorderEngine.java (excerpt)

```
  private static final String[] pageNames =
    { "Home", "Credo", "Legal" };

  public String[] getPageNames()
  {
    return pageNames;
  }
```

How did Tapestry know that the type 'Border' corresponded to the specification `/tutorial/border/Border.jwc`? Only because we defined an alias in the application specification:

Figure 6.9. Border.application (excerpt)

```
  <component-alias type="Border" specification-path="/tutorial/border/Border.jwc"/>
```

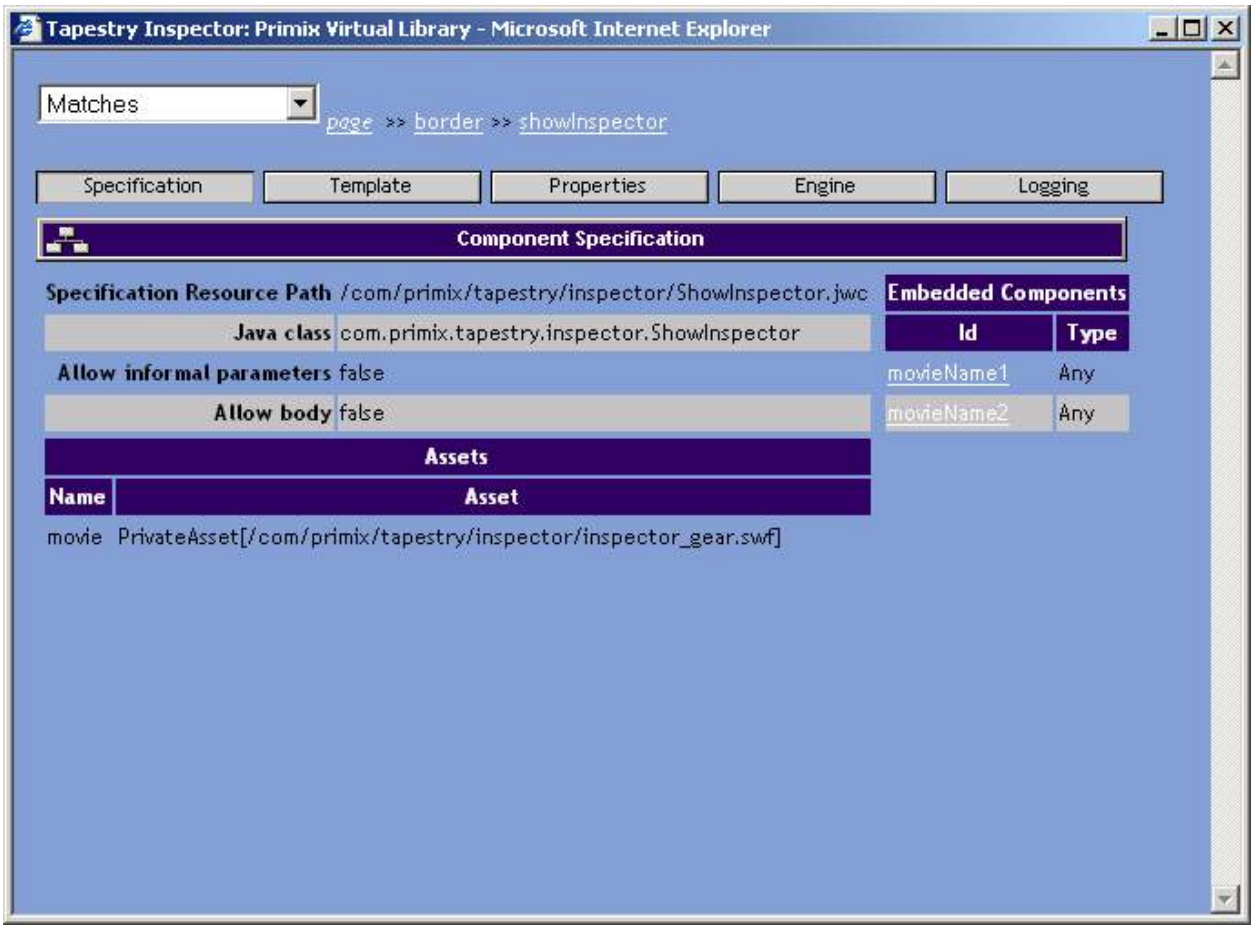
Had we failed to do this, we would have had to specify the complete resource path, `/tutorial/border/Border.jwc`, on each page's specification, instead of the short alias 'Border'. There is no magic about the existing Tapestry component types (`Insert`, `Foreach`, `PageLink`, etc. ... they each have an alias pre-registered into every application specification. These short aliases are simply a convienience.

Chapter 7. The Tapestry Inspector

Unlike scripting systems (such as JavaServer Pages and the like), Tapestry applications are gifted with a huge amount of information about how they are implemented. The same component object model that allows Tapestry to perform so many ordinary functions can be leveraged to provide some unusual functionality.

Run the Border tutorial from the previous chapter and click on the show inspector button (the gear icon in the lower right corner). A new window will launch, containing the Inspector:

Figure 7.1. Tapestry Inspector



The Inspector displays live information from the running application; in fact, it is simply another part of the application (the drop-down list of pages will include the Inspector page itself). The Inspector is most often used to debug HTML generation by viewing the HTML templates. It is also very useful in debugging problems where the wrong data is displayed, since it allows the developer to navigate to the particular components and see directly what properties are used.

Navigation

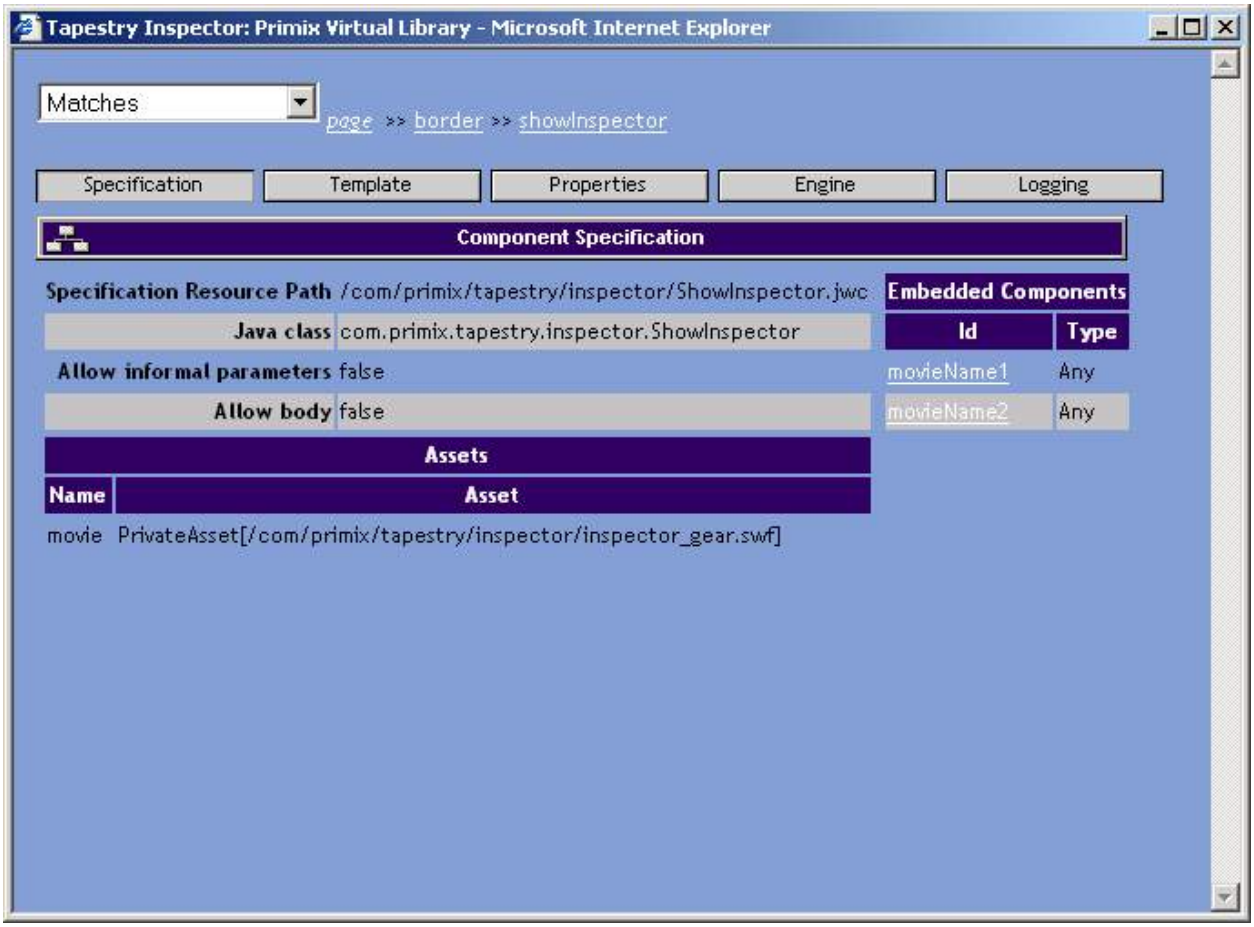
The inspector allows the user to navigate to any page and any component on a page. The drop down list in the upper left corner lists all pages in the application; changing the selection immediately updates the Inspector.

Next to the drop down list is the component path; a list of nested component ids, starting with "page" to represent the page. Clicking on any id in the path changes the information displayed below.

Underneath the component navigation tools are a set of tab buttons for the different inspector views.

Specification View

Figure 7.2. Specification View



The specification view shows several sets of information about the selected component.

First shown are basic properties, such as the specification path and Java class.

Each formal parameter is displayed. Unbound parameters will show no value in the Binding column.

Beneath formal parameters are informal parameters (the `Border` component has none, so there is nothing to see). Informal parameters are usually mapped directly to HTML attributes. They are most often used with components that generate a single HTML tag, such as the `ActionLink`, `DirectLink` or `TextField` components.

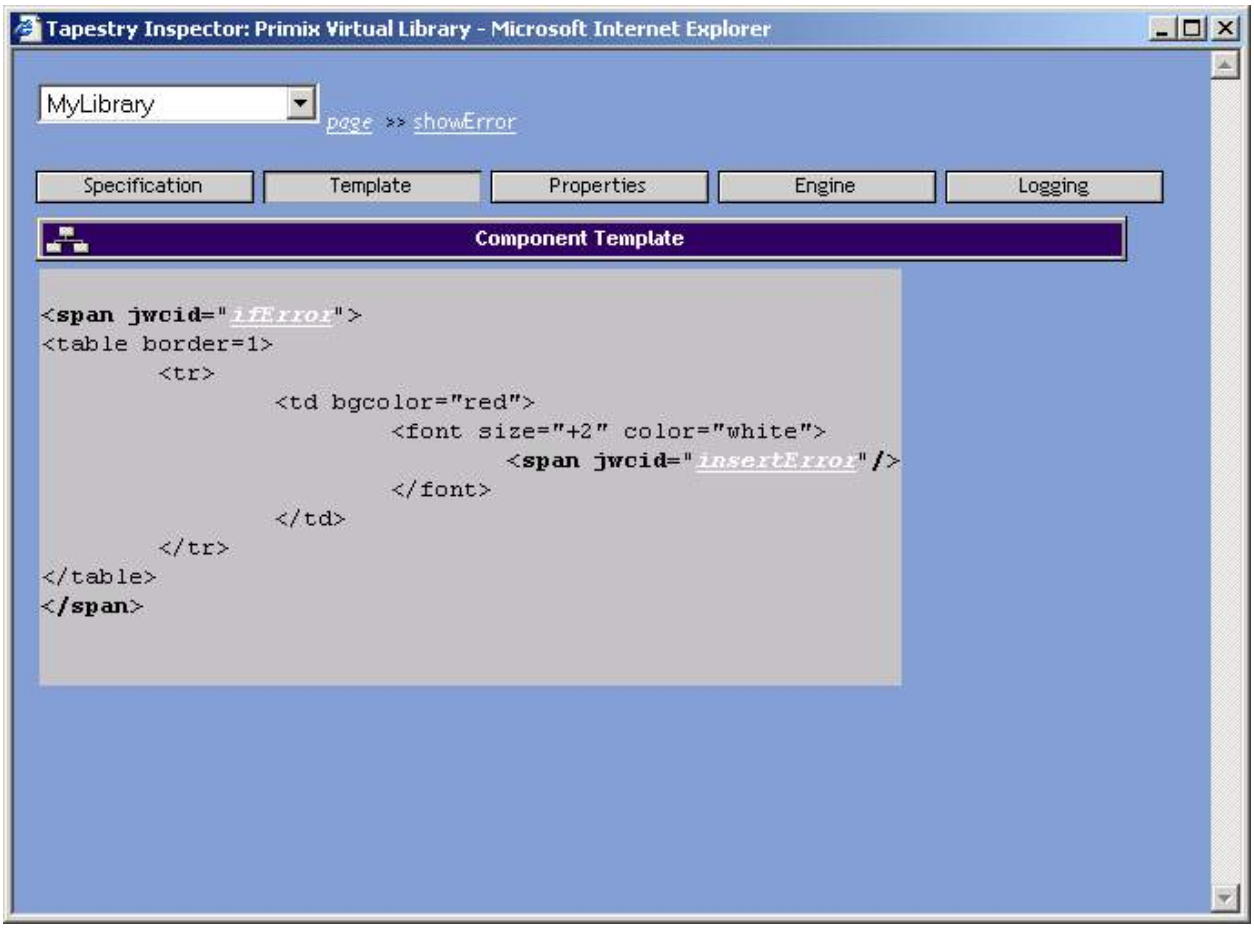
If the component contains assets, they are shown next.

Any helper beans for the component are displayed last.

On the right side is a list of each embedded component and its type. Clicking the component id will navigate to the selected component.

Template View

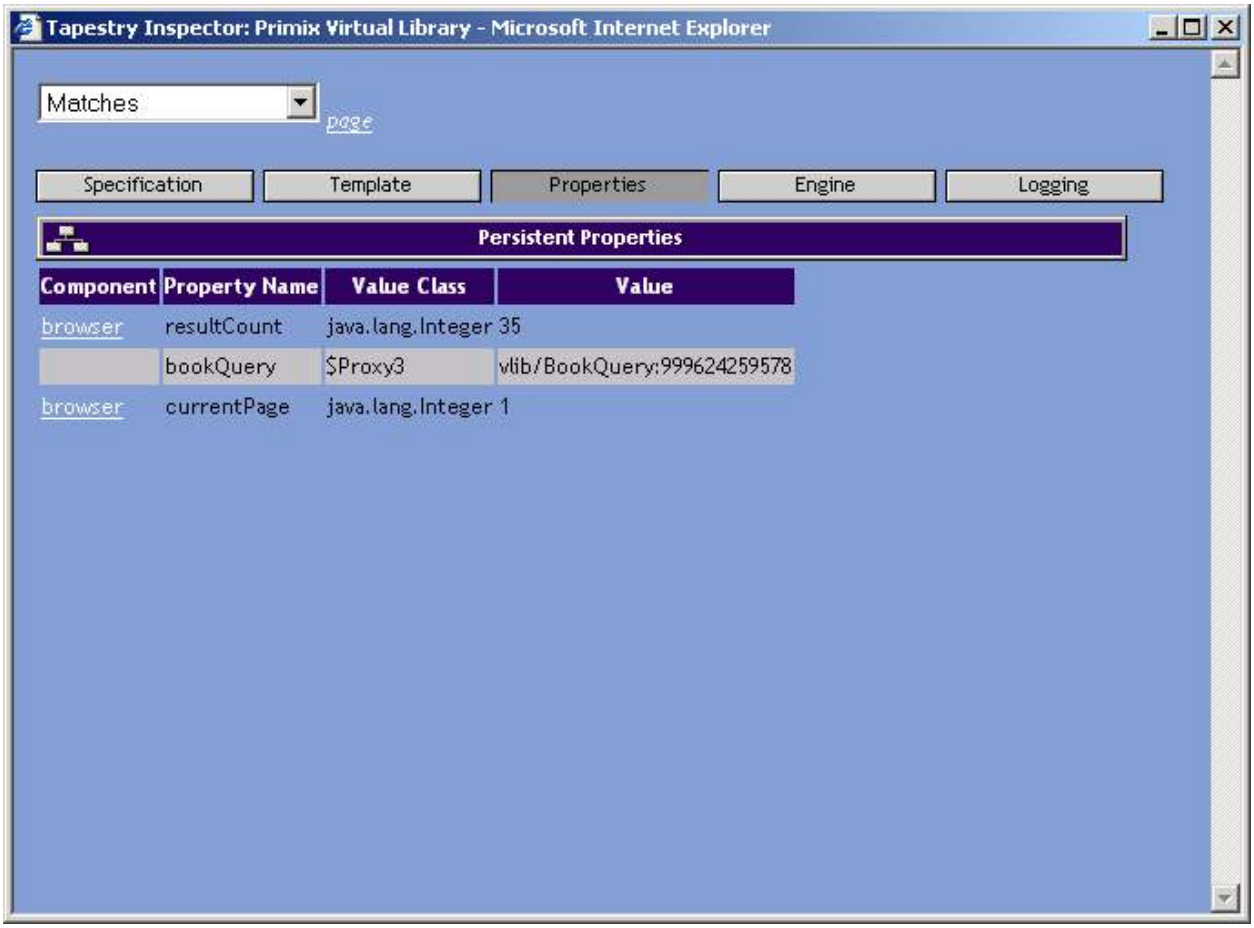
Figure 7.3. Template View



The template view shows the HTML template for the component. It shows dynamic tags in bold, and makes the component id a clickable link (which navigates to the component, but maintains the Template View). This allows the developer to quickly drill down through the components.

Properties View

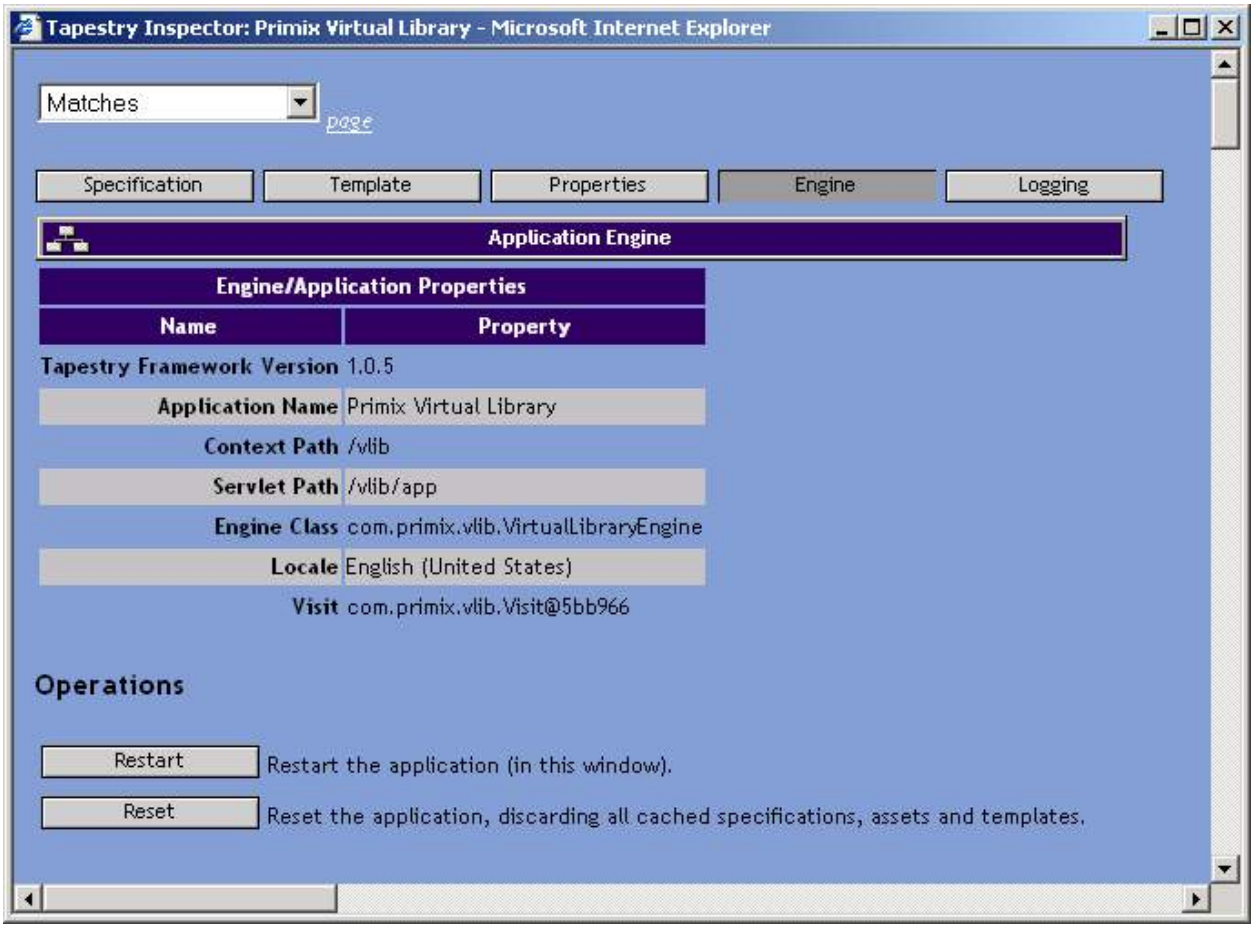
Figure 7.4. Properties View



The properties view shows persistant properties stored by the page (or any components on the page). Most pages do not store any persistent state (it is more often stored in the application's visit object).

Engine View

Figure 7.5. Engine View



The engine view shows information about the running application engine, as well as some details from the application specification.

Under Operations are two buttons: the first restarts the application. The second (when enabled [5]) resets the application, which forces a reload of all component specifications and HTML templates. This is useful during development, since it allows for incremental development without stopping and restarting the servlet container.

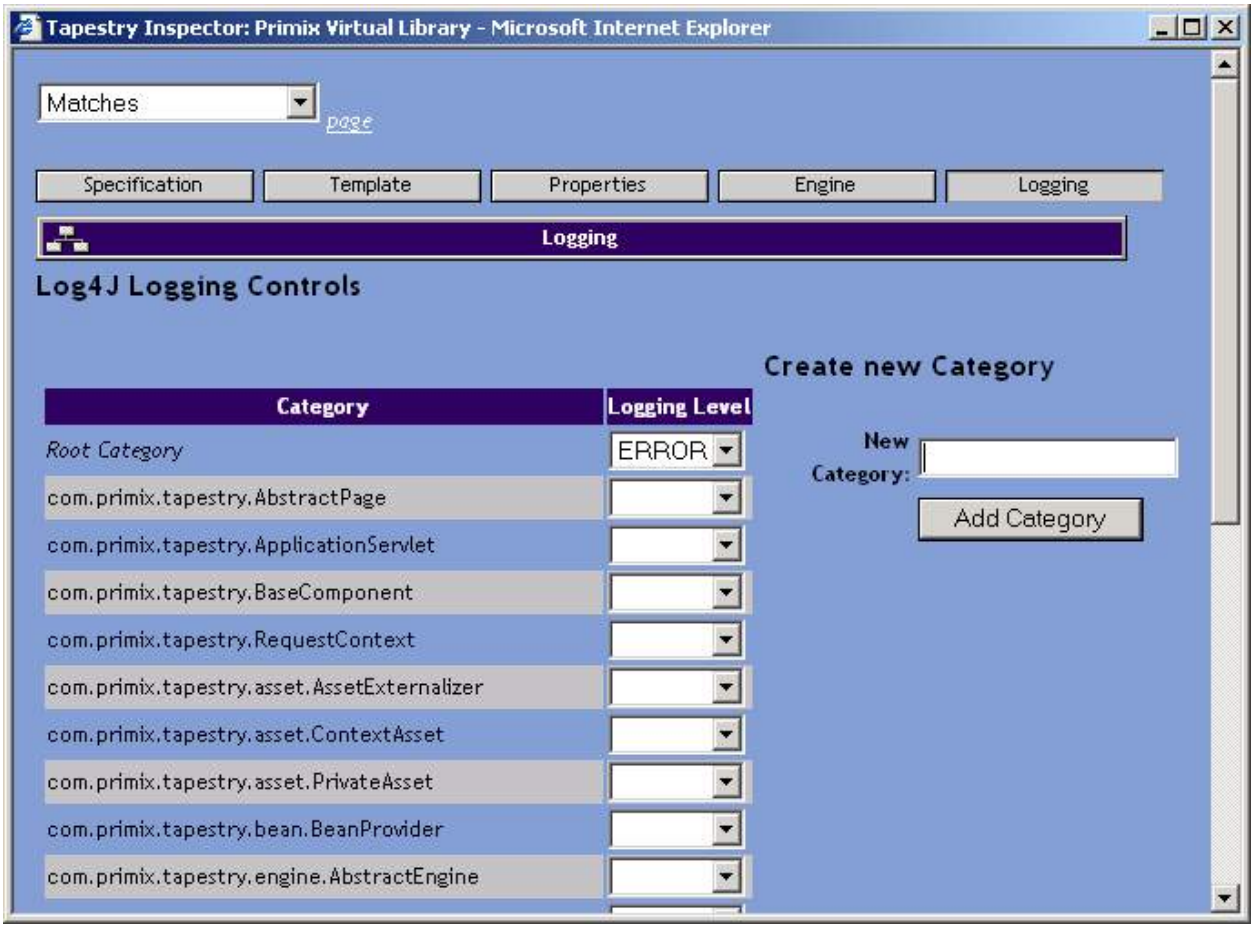
Below the operations is a binary dump of the application engine. This is useful when developing to see how large the serialized state is, and perhaps gleam how it might be trimmed.

Further below (and not visible in the screen shot above), is a dump of the request context. This is that vast amount of data also displayed when an unexpected exception is thrown.

Logging View

Figure 7.6. Logging View (Level Selection)

[5] By default, the reset service (used by the reset button) is disabled. To enable it, set the JVM system property `org.apache.tapestry.enable-reset-service` to true. The service is disabled since it is too tempting a target for a denial of service attack.



The Logging view allows dynamic integration with the Log4J logging framework. The top half of the page allows the logging level of any category to be dynamically set. This is useful when debugging, since logging output for specific classes [6] can be individually enabled or disable.

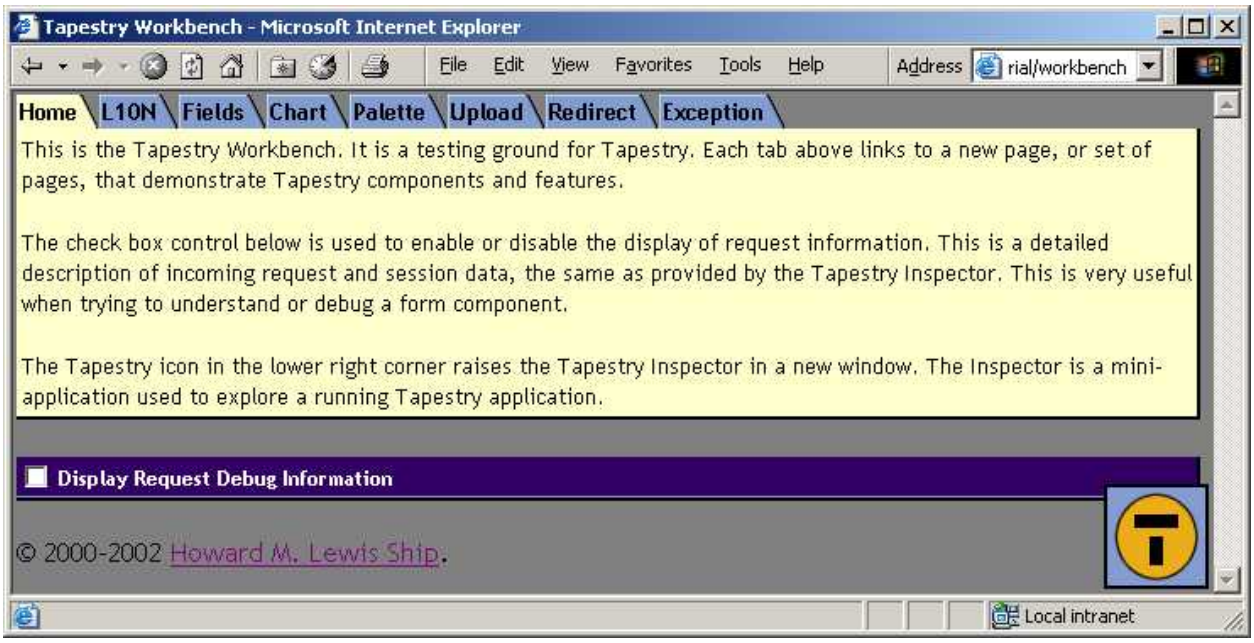
The right side is a small second form, allowing new categories to be created. This can be useful to make broad changes in logging levels. For instance, creating a category "org.apache.tapestry" would allow the logging level of all Tapestry classes to be set in a single place.

[6] By convention, logging categories match the complete class name of the corresponding class. All Tapestry logging categories conform to this convention.

Chapter 8. Tapestry Workbench

The Tapestry tutorial includes an additional application, the Workbench, which is used to show off interesting Tapestry components and features.

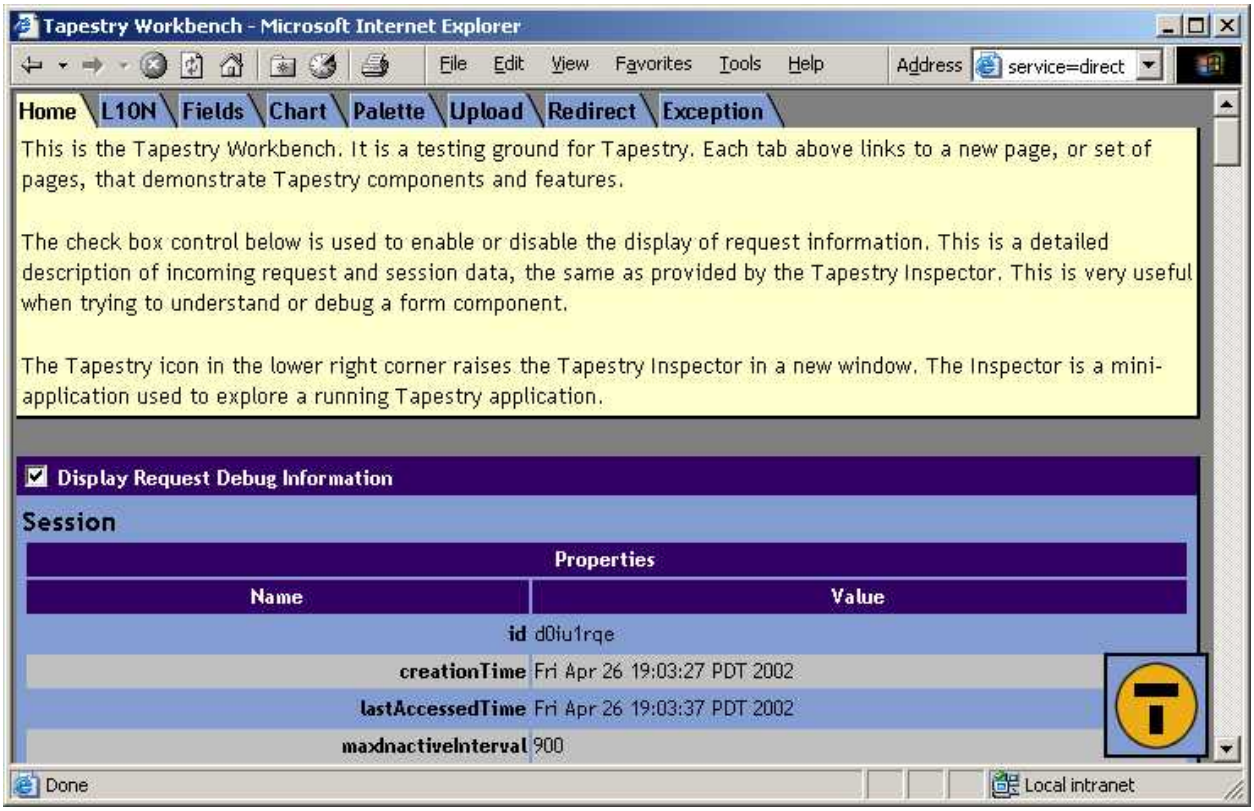
Figure 8.1. Workbench



The Workbench is divided into several areas, as shown by the tabs across the top of the page. Over time, the Workbench and this tutorial document will expand together, and the number of tabs will increase.

In addition to the Inspector, the Workbench has a useful feature which can be activated using the checkbox at the bottom. When enabled, the complete (and verbose) information available about the request, session and context (normally displayed by the Inspector's engine view) is shown at the bottom of each page.

Figure 8.2. Workbench (Showing Requests)



This feature can be very useful if you are interested in exactly how Tapestry forms and links work.

Chapter 9. Localization

One of the most powerful and useful features of the Tapestry framework is the way in which it assists with localization of a web application. This is normally an ugly area in web applications, with tremendous amounts of ad-hoc coding necessary.

Because Tapestry does such a strong job of seperating the presentation of a component (its HTML template) from its control logic (its specification and Java class) it becomes easy for it to perform localization automatically. It's as simple as providing additional localized HTML templates for the component, and letting the framework select the proper one.

However, the static text of an application, provided by the HTML templates, is not all.

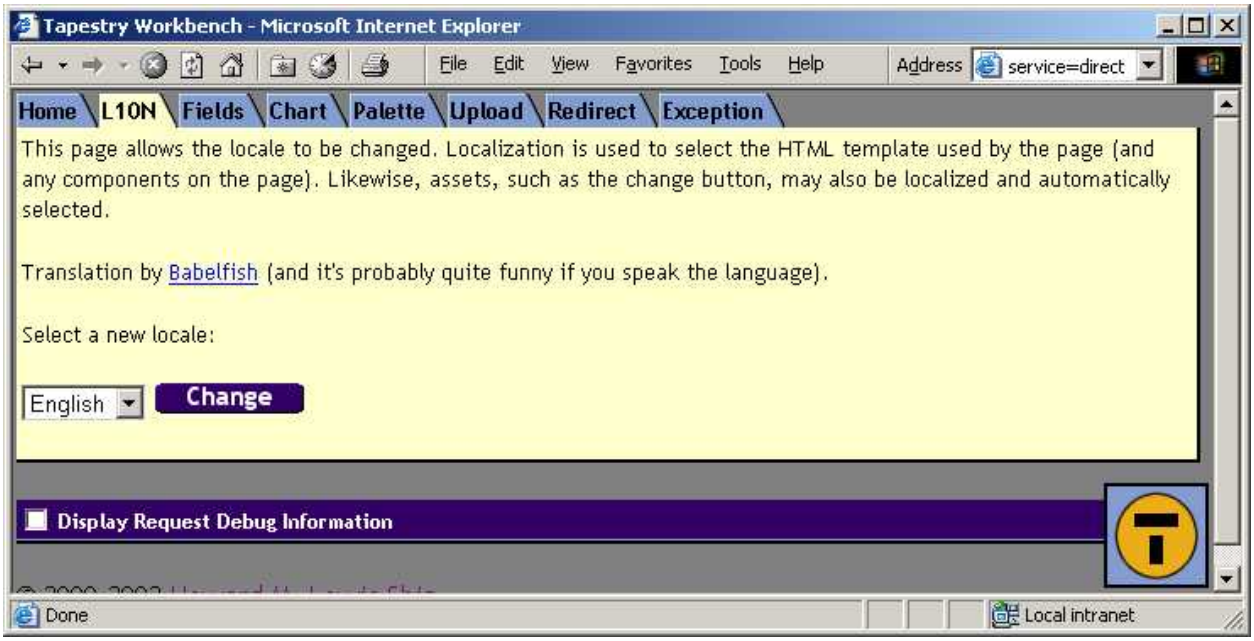
Applications also have assets (images, stylesheets and the like) that must also be localized: that fancy button labeled "Search" is fine for your English clients, but your French clients will require a similar button labeled "Recherche".

Again, the framework assists, because it can look for localized versions of the assets as it runs.

The locale application demostrates this. It is a very simply application that demonstrates changing the locale of a running application [7]

A demonstration of localization is built into the Workbench, under the L10N [8] tab. The page allows the user to select a new language for the application:

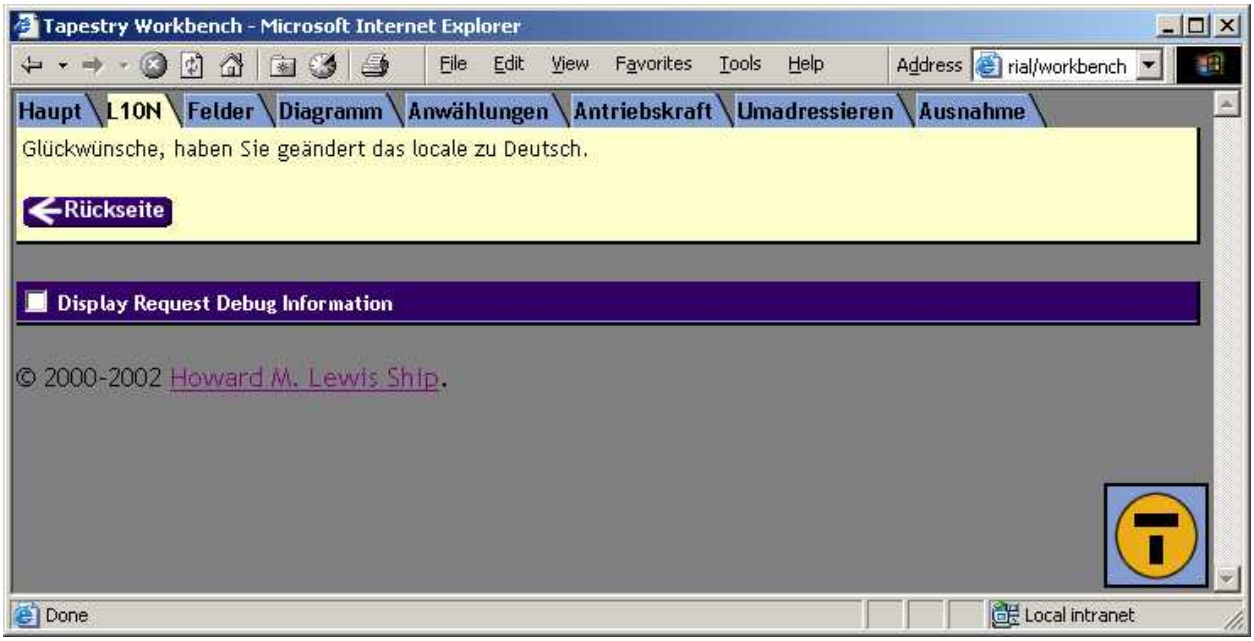
Figure 9.1. L10N Page (English)



Selecting "German" from the list and clicking the "Change" button brings you to a new page that acknowledges your selection:

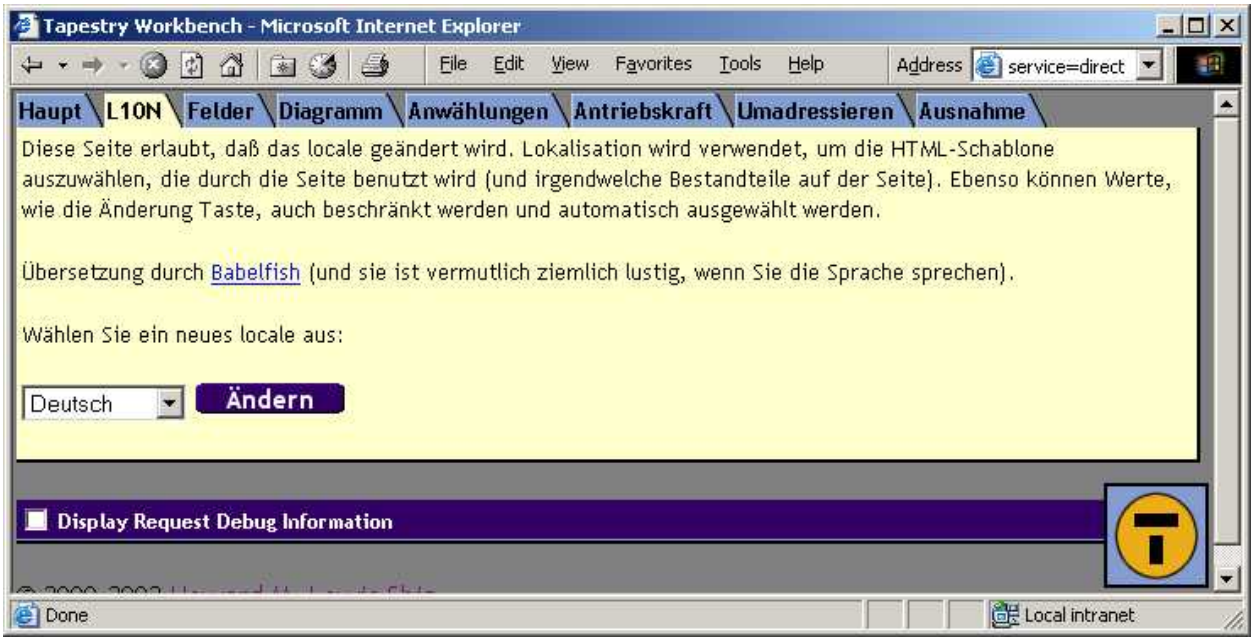
7 All the translations were performed using Babelfish, and are probably quite laughable to someone who actually speaks the alternate languages.
8 The "10" refers to the number of letters between 'l' and 'n' in the word 'localization'

Figure 9.2. Locale Changed (German)



Clicking the button (it's labeled "Return" in German) returns you to the L10N page to select a new language:

Figure 9.3. L10N Page (German)



The neat thing here is that the `L10N` page has been localized into German as well; it shows equivalent German text, the options in the popup list are in German, and the "Change" button has been replaced with a German equivalent.

Localization of HTML Templates

Localization of HTML templates ia automatic. When Tapestry reads a template, it looks for a localized version of it. In this example, in addition to the English language `Localization.html`, three additional files were created: `Localization_de.html`, `Localization_fr.html` and `Localization_it.html`.

Tapestry tracks the locale for each user using either an HTTP Cookie, or the `HttpSession`. It makes sure that all templates for all components on the page use the best available template; it does a standard search.

Localization of Assets

In the L10N pages, there are images that are also localized. Tapestry has a hand in this as well. As with HTML templates, Tapestry searches for matches based on the user's locale.

Both context assets (assets that are part of the WAR) and private assets (assets that are stored in Java frameworks) can be localized. This is demonstrated on the L10N page: the "Change" button is a private asset; the "Back" button is a context asset.

Other Options for Localization

In some cases, different localizations of the a component will be very similar, perhaps having only one or two small snippets of text that is different. In those cases, it may be easier on the developer to not localize the HTML template, but to replace the variant text with an `Insert` component.

The page can read a localized strings file (a `.properties` file) to get appropriate localized text. This saves the bother of maintaining multiple HTML templates. This is the same approach taken by the Apache Struts framework.

All components on a page share the single locale for the page, but each performs its own search for its HTML template. This means that some components may not have to be localized, if they never contain any static HTML text. This is sometimes the case for reusable components, even navigational borders.

Chapter 10. Further Study

The preceding chapters cover many of the basic aspects of Tapestry. You should be comfortable with basic Tapestry concepts:

- Seperation of presentation, business and control logic
- Use of JavaBeans properties as the source of dynamic data
- How bindings access JavaBeans properties to provide data to components
- How components wrap each other, allowing for the creation of very complicated components through aggregation
- Different types of page properties (transient, dynamic, persistent)

Tapestry is capable of quite a bit more. Also available within the Tapestry Examples package (along with the tutorial code and this document) is the Virtual Library application (Vlib).

Vlib is a full-blown J2EE application, that makes use of Tapestry as its front end, and a set of session and entity Enterprise JavaBeans as its back end.

Vlib also demonstrates some of the other aspects of developing a Tapestry application. It shows how to create pages that are bookmarkable (meaning that their URL includes enough information to reconstruct them in a subsequent session). It shows how to handle logging in to an application, and how to protect pages from being accessed until the user is logged in. It has many specialized reusable components for creating links to pages about books and people.