

PRIMIX SOLUTIONS

Core Labs

Tapestry Tutorial

CORE LABS

Tapestry Tutorial

© Primix Solutions
One Arsenal Marketplace
Phone (617) 923-6639 • Fax (617) 923-5139

Tapestry contact information:

Howard Ship <hship@primix.com>
<http://sourceforge.net/projects/tapestry>

Table of Contents

Introduction	2	Stale Sessions	26
Setting up the Tutorial	3	Exception Handling	27
Java Build Environment (JBE)	3	Re-usable Components	30
Building the Tutorial	3	The Tapestry Inspector	37
Hello World	7	Specification View	38
Application Object	7	Components View	39
Application Servlet	8	Template View	40
Application Specification	8	Properties View	41
Home Page Specification	9	Complex Forms and Output	43
Home Page Template	9	Survey	45
Run the Application	10	SurveyDatabase	49
Dynamic Content	11	SurveyApplication	49
Interactive Application	16	SurveyPage	50
Adding Interactivity using Listeners	23	Results	58
Persistent Page State and Page		Further Study	66
Pooling	24		
Dynamic Page State	25		
Tapestry Run Time Errors	26		



Introduction

Tapestry is a new application framework developed at Primix Solutions.

Tapestry 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 at 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 session, 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.

Setting up the Tutorial

Before we can get started writing servlets, we must have an environment for running them.

This document expects that you will have extracted the Tapestry distribution to your `c:` drive. This will have created a directory `Tapestry-x.x.x1` and, beneath it, several more directories.

The Tapestry Tutorial will be in `C:\Tapestry-x.x.x\Examples\Tutorial`.

Java Build Environment (JBE)

In order to execute the tutorial, you must set up the **Java Build Environment** or JBE, which is also distributed with Tapestry. The file `C:\Tapestry-x-x-x\JBE\Manual.pdf` contains a full description of how to set up the JBE.

Setting up the JBE is required, since it is used to compile the tutorial as well as execute it.

Building the Tutorial

Building the Tutorial is quite easy (once the JBE is set up, and the necessary Jar files installed).

Change to the Tutorial directory, `C:\Tapestry-x-x-x\Examples\Tutorial`.

Execute the command `make war`

```
C:\Tapestry-0.2.5\Examples\Tutorial>make war
*** Cataloging package tutorial.hello ... ***

*** Cataloging package tutorial.simple ... ***
```

¹The actual release numbers will change. This document was prepared for Tapestry-0-2-6.

```

*** Cataloging package tutorial.adder ... ***

*** Cataloging package tutorial.border ... ***

*** Cataloging package tutorial.survey ... ***

*** Compiling ... ***

cd . ; \
C:/jdk1.2.2/bin/javac.exe -d C:/Tapestry-
0.2.5/Examples/Tutorial/.build/wapp/WEB
-INF/classes -classpath "C:/Tapestry-0.2.5/Examples/Tutorial;C:/Tapestry-
0.2.5/E
xamples/Tutorial/.build/wapp/WEB-INF/classes;C:/Tapestry-
0.2.5/lib/Tapestry.jar;
C:/Tapestry-0.2.5/lib/javax.servlet.jar"
tutorial/hello/HelloWorldServlet.java t
utorial/simple/Home.java tutorial/simple/SimpleServlet.java
tutorial/adder/Adder
Servlet.java tutorial/adder/Home.java tutorial/border/Border.java
tutorial/borde
r/BorderApplication.java tutorial/border/BorderServlet.java
tutorial/survey/Race
.java tutorial/survey/RaceModel.java tutorial/survey/Results.java
tutorial/surve
y/Sex.java tutorial/survey/SexAdaptor.java tutorial/survey/Survey.java
tutorial/
survey/SurveyApplication.java tutorial/survey/SurveyDatabase.java
tutorial/surve
y/SurveyPage.java tutorial/survey/SurveyServlet.java

*** Copying WEB-INF resources ... ***

Copying: web.xml

*** Copying package resources ...***

Copying: HelloWorld.application Home.html Home.jwc Simple.application
Home.html
Home.jwc Adder.application Home.html Home.jwc Border.application
Border.html Cre
do.html Home.html Legal.html Border.jwc Credo.jwc Home.jwc Legal.jwc
Survey.appl
ication Home.html Results.html SurveyPage.html Home.jwc Results.jwc
SurveyPage.j
wc SurveyStrings.properties

*** Copying context resources ... ***

Copying: index.html

*** Building Tutorial.war ... ***

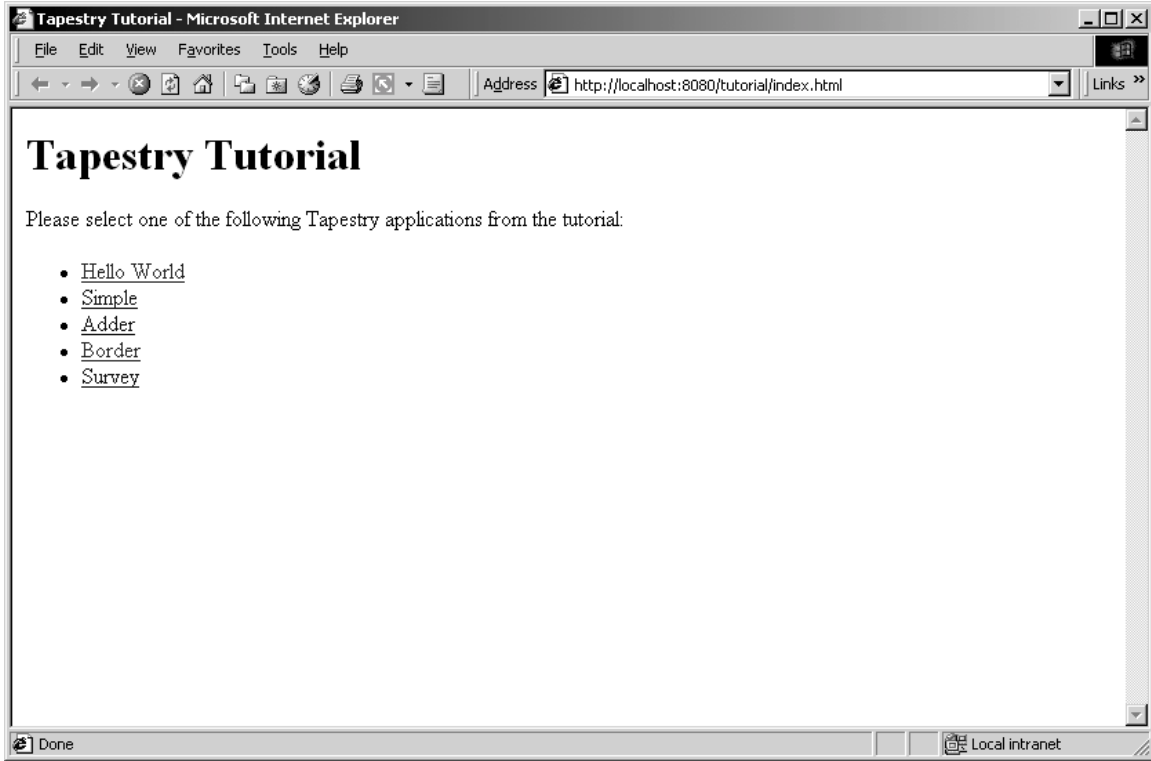
```

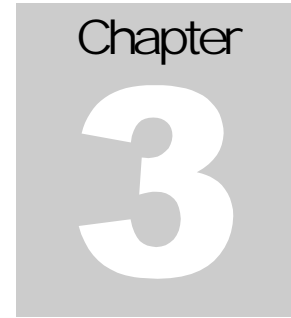
```
C:/jdk1.2.2/bin/jar.exe cf Tutorial.war -C .build/wapp .
C:\Tapestry-0.2.5\Examples\Tutorial>
```

That's an awful amount of output. When its done, you can start up the Jetty server just as easily with the `make run` command:

```
C:\Tapestry-0.2.5\Examples\Tutorial>make run
C:/jdk1.2.2/bin/java.exe -classpath "C:/Tapestry-
0.2.5/lib/Tapestry.jar;C:/Tapes
try-0.2.5/lib/javax.servlet.jar;C:/Tapestry-
0.2.5/lib/xerces.jar;C:/Tapestry-0.2
.5/lib/gnu-regexp.jar;C:/Tapestry-0.2.5/lib/com.mortbay.jetty.jar" \
-Dorg.xml.sax.parser=org.apache.xerces.parsers.SAXParser \
com.mortbay.Jetty.Server jetty.xml
20001109221653438GMT EVENT
[main]com.mortbay.HTTP.HttpServer.addWebApplicatio
n(HTTPServer.java:480)
  >>9> Web Application WebApp:Tapestry Tutorial@jar:file:/C:/Tapestry-
0.2.5/Exam
ples/Tutorial/Tutorial.war!/ added
20001109221653438GMT EVENT
[main]com.mortbay.HTTP.Handler.NullHandler.start(N
ullHandler.java:79)
  >>5> Started ServletHandler in WebApp:Tapestry
Tutorial@jar:file:/C:/Tapestry-
0.2.5/Examples/Tutorial/Tutorial.war!/
20001109221653438GMT EVENT
[main]com.mortbay.HTTP.Handler.ResourceHandler.sta
rt(ResourceHandler.java:151)
  >>4> ResourceHandler started in jar:file:/C:/Tapestry-
0.2.5/Examples/Tutorial/
Tutorial.war!/
20001109221653438GMT EVENT
[main]com.mortbay.HTTP.Handler.NullHandler.start(N
ullHandler.java:79)
  >>5> Started ResourceHandler in WebApp:Tapestry
Tutorial@jar:file:/C:/Tapestry
-0.2.5/Examples/Tutorial/Tutorial.war!/
20001109221653438GMT EVENT
[main]com.mortbay.HTTP.SocketListener.start(Socket
Listener.java:71)
  >>3> Started SocketListener on 0.0.0.0/0.0.0.0:8080
```

Finally, you can access the Tutorials using the URL <http://localhost:8080/tutorial>





Hello World

We will develop a very simple, completely static web application as an introduction to the basic concepts of Tapestry.

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.

Even this simple Tapestry application requires two objects:

- An application object that runs our (very simple) application
- A servlet that bridges between the servlet container and our application

After that, 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-0.2.5\Examples\Tutorial\tutorial\hello`.

Application Object

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

The application object is an instance, or subclass of, the Tapestry class `com.primix.tapestry.app.SimpleApplication`.

In these first few examples, we have no additional behavior to add to the provided base class, so we simply use `SimpleApplication` as our application's class.

Application Servlet

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

The application servlet must subclass `com.primix.tapestry.ApplicationServlet` and implement two methods: `getApplicationSpecificationPath()` and `createApplication()`. The first method provides the path to the application specification file; the servlet reads this file when it is initialized.

The second method creates the application instance (on the first request from the client). Tapestry will store the application instance into the `HttpSession` so that it will be available on subsequent requests from the client.

HelloWorldServlet.java

```
package tutorial.hello;

import com.primix.tapestry.*;
import com.primix.tapestry.app.*;

public class HelloWorldServlet extends ApplicationServlet
{
    protected String getApplicationSpecificationPath()
    {
        return "/tutorial/hello/HelloWorld.application";
    }

    protected IApplication createApplication(RequestContext context)
    {
        return new SimpleApplication(context, null);
    }
}
```

Application Specification

The application specification is used to describe the application to the Tapestry framework. It provides the application with a name, 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: either in a Jar file, or in the `WEB-INF/classes` directory of a war (Web Application Archive).

HelloWorld.application

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC
    "-//Primix Solutions//Tapestry Specification 1.0//EN"
    "http://tapestry.sourceforge.net/dtd/Tapestry_1_0.dtd">

<application>
  <name>Hello World Tutorial</name>
```

```

    <page>
      <name>Home</name>
      <specification-path>/tutorial/hello/Home.jwc
      </specification-path>
    </page>
  </application>

```

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

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:

```

Home.jwc
<?xml version="1.0"?>
<!DOCTYPE specification PUBLIC
  "-//Primix Solutions//Tapestry Specification 1.0//EN"
  "http://tapestry.sourceforge.net/dtd/Tapestry_1_0.dtd">

<specification>
  <class>com.primix.tapestry.BasePage</class>
</specification>

```

This simply says that Home is a kind of page. We use the supplied Tapestry class `com.primix.tapestry.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".

```

Home.html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

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

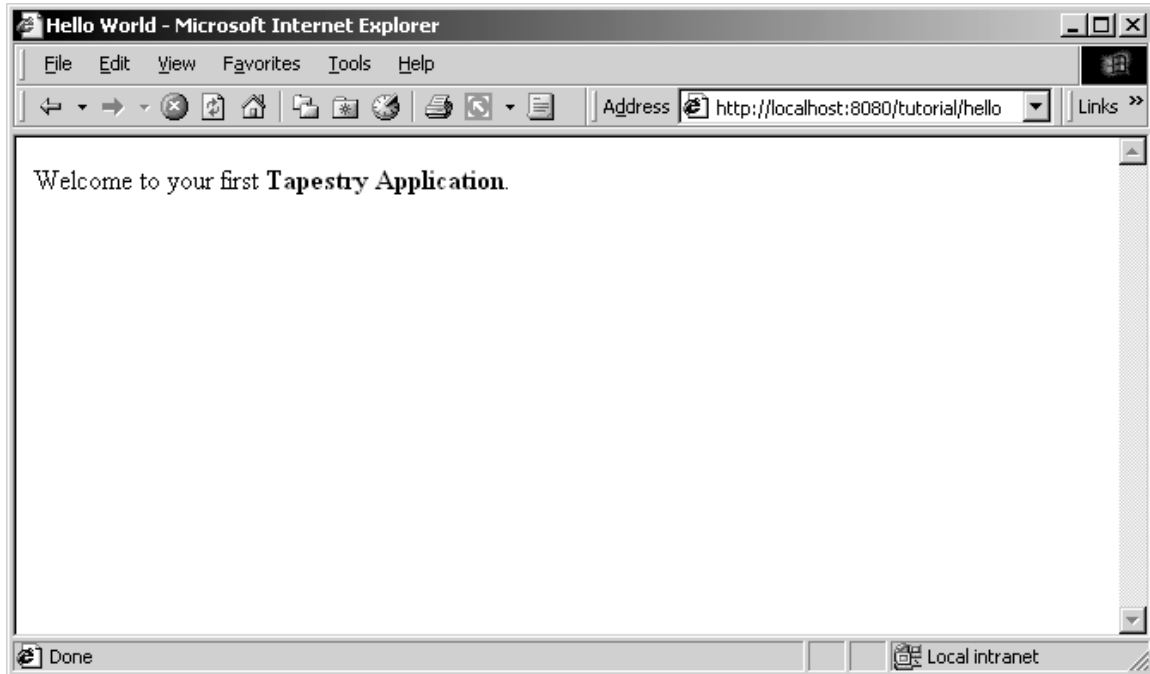
<body>

```

```
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:



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. There was an application consisting of a single component.

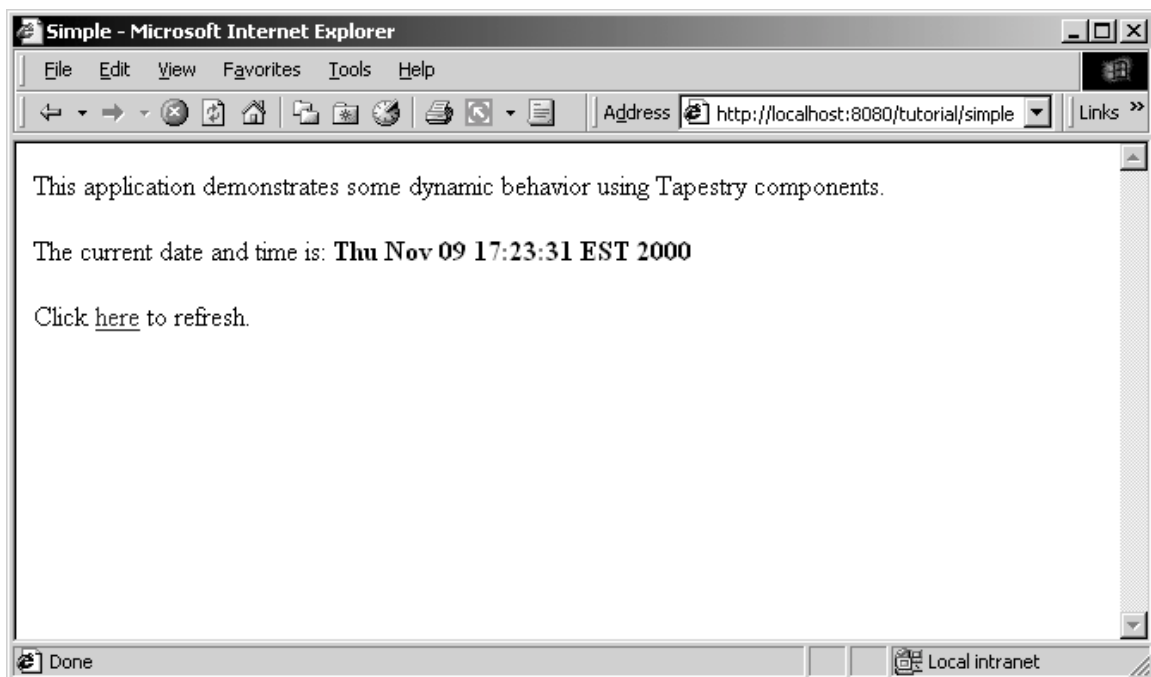
In the following chapters, we'll see how to add dynamic content and then true interactivity.

Dynamic Content

This example will add a tiny amount of interactivity, as well as some very simple dynamic content ... content that is different each time the 'page' is viewed.

In this section, 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:



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`.

We need to create a new servlet and application object, but they're almost identical to our earlier ones (only the parts marked in blue are different). The real action in this section will be the new version of the home page.

SimpleServlet.java

```
package tutorial.simple;

import com.primix.tapestry.*;
import com.primix.tapestry.app.*;

public class SimpleServlet extends ApplicationServlet
{
    protected String getApplicationSpecificationPath()
    {
        return "/tutorial/simple/Simple.application";
    }

    protected IApplication createApplication(RequestContext context)
    {
        return new SimpleApplication(context, null);
    }
}
```

The bold text identifies the only significant changes from the previous HelloWorldServlet class.

The application specification is also straight forward:

Simple.application

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC
    "-//Primix Solutions//Tapestry Specification 1.0//EN"
    "http://tapestry.sourceforge.net/dtd/Tapestry_1_0.dtd">

<application>
    <name>Simple Tutorial</name>

    <page>
        <name>Home</name>
        <specification-path>/tutorial/simple/Home.jwc
        </specification-path>
    </page>
</application>
```

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

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><jwc id="insertDate"/></b>

<p>Click <jwc id="refresh">here</jwc> to refresh.

</body>
</html>

```

This looks like ordinary HTML, except for the special `<jwc>` tags (shown in bold). "jwc" is an abbreviation for "Java Web Component"; these tags are placeholders for the dynamic content provided by Tapestry components.

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

One of the goals of Tapestry is that the HTML should have the minimum amount of special markup. This is demonstrated here ... the `<jwc>` 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, the particular type and configuration of each component is defined in the component specification.

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 `` 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 `<jwc>` tags) 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.

Home.jwc

```

<?xml version="1.0"?>
<!DOCTYPE specification PUBLIC
    "-//Primix Solutions//Tapestry Specification 1.0//EN"

```

```

"http://tapestry.sourceforge.net/dtd/Tapestry_1_0.dtd">
<specification>
  <class>tutorial.simple.Home</class>

  <components>
    <component>
      <id>insertDate</id>
      <type>Insert</type>

      <bindings>
        <binding>
          <name>value</name>
          <property-path>currentDate</property-path>
        </binding>
      </bindings>
    </component>

    <component>
      <id>refresh</id>
      <type>Page</type>

      <bindings>
        <static-binding>
          <name>page</name>
          <value>Home</value>
        </static-binding>
      </bindings>
    </component>
  </components>
</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 `Page`, meaning it creates a link to some other page in the application. `Page` components have a parameter, also 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.

That just leaves the implementation of the Home page component:

Home.java

```
package tutorial.simple;

import java.util.*;
import com.primix.tapestry.*;

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 Tapestry:

```
<!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>Thu Nov 09 17:23:31 EST 2000</b>

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

</body>
</html>
```

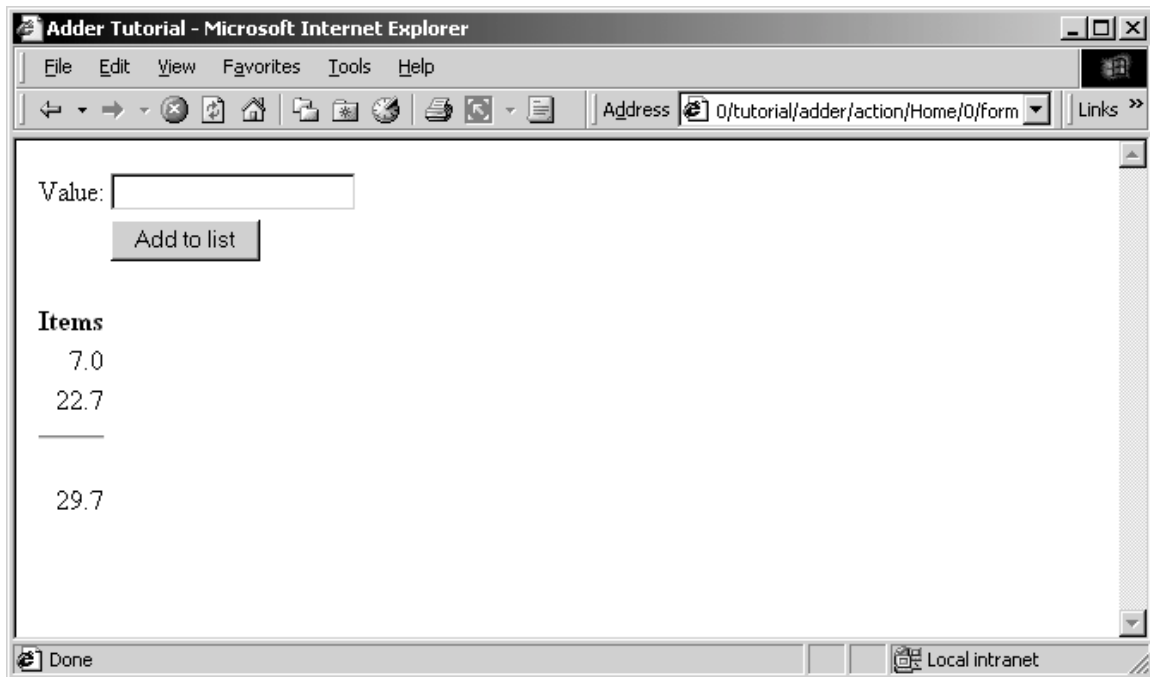
This should look very familiar. Text which was generated dynamically, by Tapestry components, is in bold font. As you can see, Tapestry not only inserted simple text (the current date and time, obtained from an `java.util.Date` object), but the refresh component inserted the `<a>` and `` tags, and created an appropriate URL for the href attribute.

Interactive Application

A more ambitious example, we'll build a simple adding machine.

Now it's time to build a real, interactive application. We'll still use just a single page, but it will demonstrate many of the more interesting features of Tapestry, including maintenance of server side page state.

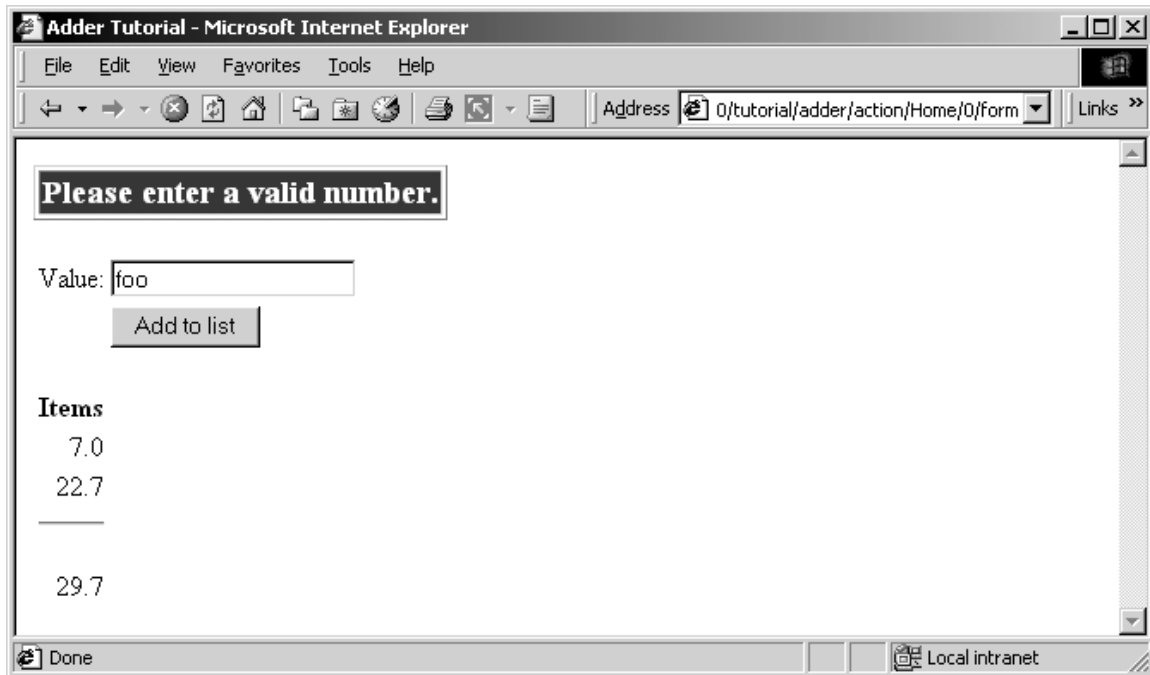
Our Adder application allows the user to sum up a list of numbers.



The user enters a number into the value field and clicks "Add to list". The number is added to the list of items and factored into the total.

A Form component containing a TextField component will be used to collect information from the user. A Foreach component will be used to run through the list of items, and Insert components will be used to present each item in the list, as well as the total.

If the user enter in a non-number, then an error message is displayed.



As with the previous examples, the servlet and application objects are simple variations on the previous two sets (they are omitted here).

The application specification is, likewise, a variation on the prior example.

The code for this section is in the tutorial.adder package.

We'll start with the HTML template for the home page:

```

Home.html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<html>
<head>
  <title>Adder Tutorial</title>
</head>
<body>

<jwc id="ifError">
<table border=1>
  <tr>
    <td bgcolor=red>
      <span style="font: bolder 14pt; color:white">
        <jwc id="insertError"/>
      </span>
    </td>
  </tr>
</table>
<p>
</jwc>

```

```

<jwc id="form">
<table>
  <tr>
    <td align=right>Value:</td>
    <td><jwc id="inputNewValue"/></td>
  </tr>
  <tr>
    <td> </td>
    <td><input type=submit value="Add to list"></td>
  </tr>
</table>

</jwc>

<table>
  <tr> <th>Items</th> </tr>
<jwc id="e">
  <tr align=right>
    <td>
      <jwc id="insertCurrentValue"/>
    </td>
  </tr>
</jwc>

  <tr align=right>
    <td>
      <hr>
      <br><jwc id="insertTotal"/>
    </td>
  </tr>
</table>

</body>
</html>

```

Again, Tapestry takes care of most of the details. The form component will turn into an HTML `<FORM>` element, and the correct URL is automatically generated. The textfield component will become an `<INPUT TYPE=TEXT>`, with the necessary smarts to collect the value submitted by the user and provide it to the page.

The `e` component is a `Foreach`, used for running through a list of elements (supplied as a `List`, `Iterator` or an array of Java objects). We've already see the `Insert` component.

Next we have the specification:

Home.jwc

```

<?xml version="1.0"?>
<!DOCTYPE specification PUBLIC "-//Primix Solutions//Tapestry Specification
1.0//EN"
  "http://tapestry.sourceforge.net/dtd/Tapestry_1_0.dtd">

<specification>
  <class>tutorial.adder.Home</class>

```

```

<components>

  <component>
    <id>ifError</id>
    <type>Conditional</type>

    <bindings>
      <binding>
        <name>condition</name>
        <property-path>error</property-path>
      </binding>
    </bindings>
  </component>

  <component>
    <id>insertError</id>
    <type>Insert</type>

    <bindings>
      <binding>
        <name>value</name>
        <property-path>error</property-path>
      </binding>
    </bindings>
  </component>

  <component>
    <id>form</id>
    <type>Form</type>

    <bindings>
      <binding>
        <name>listener</name>
        <property-path>formListener</property-path>
      </binding>
    </bindings>
  </component>

  <component>
    <id>inputNewValue</id>
    <type>TextField</type>

    <bindings>
      <binding>
        <name>text</name>
        <property-path>newValue</property-path>
      </binding>
    </bindings>
  </component>

  <component>
    <id>e</id>
    <type>Foreach</type>

    <bindings>
      <binding>
        <name>source</name>

```

```

        <property-path>items</property-path>
    </binding>
</bindings>
</component>

<component>
    <id>insertCurrentValue</id>
    <type>Insert</type>

    <bindings>
        <binding>
            <name>value</name>
            <property-path>components.e.value</property-path>
        </binding>
    </bindings>
</component>

<component>
    <id>insertTotal</id>
    <type>Insert</type>

    <bindings>
        <binding>
            <name>value</name>
            <property-path>total</property-path>
        </binding>
    </bindings>
</component>
</components>
</specification>

```

We only want to display the error message if there is one, so the `ifText` is conditional on there being a non-null error message (the `Conditional` component treats null as false).

For the form component, all we have to do is supply a listener, an object that is informed when the form is submitted.

For the `textfield` component, we provide a `text` parameter that provides the default value for the `<INPUT>` element, as well as a place to put the value submitted on the form. This must be of type `java.lang.String`, so we need to do a little translation (in our Java class), since internally we want to store the value as a double.

For the `e` component, we supply a binding for the `source` parameter. For each item in the source list, it will update its `value` property, which is later accessed by the `Insert` component. The property path `components.e.value` accomplishes this: the page has a `component` property, which is a `Map` of the components on the page. `e` is the id of a component, and a key in the `Map`. It has a property named `value`, which is the current item from the source list.

A `Foreach` also has a parameter named `value`. By creating a binding for this parameter, the `Foreach` can update a property of the page, or some other component. This is more commonly used when the items in the list are business objects and the application needs to invoke business methods on them.

Finally, the Java code for the home page puts everything together:

Home.java

```
package tutorial.adder;

import com.primix.tapestry.*;
import com.primix.tapestry.components.*;
import java.util.*;

public class Home extends BasePage
{
    private List items;
    private String newValue;
    private String error;

    public List getItems()
    {
        return items;
    }

    public void setItems(List value)
    {
        items = value;

        fireObservedChange("items", value);
    }

    public void setNewValue(String value)
    {
        newValue = value;
    }

    public String getNewValue()
    {
        return newValue;
    }

    public void detachFromApplication()
    {
        items = null;
        newValue = null;
        error = null;

        super.detachFromApplication();
    }

    public void addItem(double value)
    {
        if (items == null)
        {
            items = new ArrayList();
            fireObservedChange("items", items);
        }

        items.add(new Double(value));
    }
}
```

```

        fireObservedChange();
    }

    public double getTotal()
    {
        Iterator i;
        Double item;
        double result = 0.0;

        if (items != null)
        {
            i = items.iterator();
            while (i.hasNext())
            {
                item = (Double)i.next();
                result += item.doubleValue();
            }
        }

        return result;
    }

    public IActionListener getFormListener()
    {
        return new IActionListener()
        {
            public void actionTriggered(IComponent component,
IRequestCycle cycle)
            {
                try
                {
                    double item = Double.parseDouble(newValue);
                    addItem(item);

                    newValue = null;
                }
                catch (NumberFormatException e)
                {
                    error = "Please enter a valid number.";
                }
            }
        };
    }

    public String getError()
    {
        return error;
    }
}

```

That may seem like a lot of code for what we're doing, but in reality, very much is going that we don't have to write:

- Processing the submitted form

- Storing the List of items persistently between request cycles
- Encoding and decoding URLs
- Very robust exception support

Tapestry components, using JavaBeans properties, take care of moving data to and from the HTML form. Our application merely has to supply the logic to properly respond when the form is submitted. In this case, converting the text into a double that can be added to the list.

Because we let Tapestry set the names of our form elements, there's no possibility of mismatched names between the Java code (setting defaults and interpreting the posted request) and the HTML template.

Enter a few values into the text field to see how the application works, adding them together into an ever larger list.

Adding Interactivity using Listeners

To understand the relationship between the home page specification, the home page class and the components used by the home page, it is necessary to understand the JavaBeans properties provided by the home page class.

We implement several JavaBeans properties on this page:

Property name	Type	R / W	Description
newItem	String	R / W	Stores the string entered into the form.
items	List (of Double)	R / W	Items in the list. Persists between request cycles.
formListener	IActionListener	Read Only	Informed when form is submitted.
total	double	Read Only	Total of items; computed on the fly.

This example demonstrates how to provide interactivity to an application. For Tapestry, interactivity is defined as a request cycle initiated by a user clicking on a hyperlink or submitting a form.

In our case, we want to know when the form containing the TextField is submitted so that we can provide application specific behavior -- adding the value entered in the TextField to the list of items.

This is accomplished using a listener, an object that implements the Java interface `IActionListener`. This interface defines a single method, `actionTriggered()`. When the form is submitted, all the components wrapped by the form (in this case, the `TextField`) are given a chance to retrieve their values from the request and update properties of the application (the `TextField` sets the `currentItem` property). The form then gets its listener and invokes the `actionTriggered()` method.

In the specification, the listener parameter was bound to the `formListener` property of the page. The code in the `getFormListener()` method creates an anonymous inner class and returns it.

Inner classes have access to the private fields and methods of the class. In this case, the inner class invokes the `addItem()` method to add the `currentItem` (with a value provided by the `TextField` component) to the `items` List.

A listener is free to do anything it wants. It can change the state of the application, or can retrieve other pages (by name) from the request cycle object, and can change properties of those pages. It can even chose a different page to render, by invoking `setPage()` on the request cycle.

Persistent Page State and Page Pooling

The home page of this application uses a persistent page property, a `List` that contains `java.lang.Doubles`, the items in the list.

Persistent page state is one of the most important concepts in Tapestry. Each page in the application (and in fact, even components within the page) has some properties that should persist between requests. This can be values such as the user's name and address, or (in this case) the list of numbers entered so far.

In traditional JavaServer Pages or servlet applications, a good chunk of code must be written to manage this. The values must be encoded in cookies, as hidden form fields, as named attributes of the `HttpSession`, or stored into a server-side flat file or database. Each servle (or JSP) is directly responsible for managing access to these values ... which leads to many half realized, ad-hoc solutions and an avalanche of bugs, and even security holes.

With Tapestry, the framework takes care of these persistence issues. When a persistent property of a page is changed the accessor method also invokes the method `fireObservedChange()`. This method informs a special object, the page's recorder, about the property and its new value.

When the page is next used, the value is restored automatically. This may not seem natural ... an obvious question is: why wasn't the page in the same state? Then answer is that that page instance is shared, and may be used by a different client in the interrum.

Within the Tapestry framework, all of these pages, components, specifications and templates are converted into Java objects. Assembling a page is somewhat expensive: it involves reading all

those specifications and templates², creating and initializing component objects, creating binding objects for the components, and organizing the components into a hierarchy.

Creating a page object for just one request cycle only to discard it is simply unacceptable. Pages should be kept around as long as they are needed; they should be re-used in subsequent request cycles, both for the same client session, or for other sessions.

The Tapestry framework accomplishes this by pooling instances of page objects; there could conceivably be a handful of different instances being shared by thousands of client sessions. This is a kind of shell game that is important to maintain scalability.

What this means for the developer is some minor extra work. On each request cycle, a different instance of the page object may be used to handle the request. This means that data can't simply be stored in the instance variables of the page between request cycles.

Tapestry separates the persistent state of a page from the actual page objects. The state is stored separately, making use of the page recorder objects. When needed, a page can be created or reclaimed from the page pool and have all of its persistent properties set by the page recorder.

The developer has three responsibilities when coding a page with persistent state:

- The property must be serializable; this includes Java scalar types (boolean, int, double, etc.), Strings, common collection classes (ArrayList, HashMap, etc.) and other classes that implement `java.io.Serializable`, such as `javax.ejb.Handle`.
- When the value of the property changes, the `fireObservedChange()` method must be invoked, to inform the page recorder about the change.
- When the request cycle ends and the page is returned to the pool, the persistent state must be reset to its initial value (as if the page object was newly instantiated). This is done in the `detachFromApplication()` method.

Dynamic Page State

This page has a bit of dynamic state; state that changes as the page is being rendered. The value property of the Foreach component takes on different values from the items List as the page is rendered. Dynamic state is easier to handle than persistent state; for completeness, it must also be reset in the `detachFromApplication()` method.

² Specifications and templates are generally read just once, then left in memory for subsequent use.

Tapestry Run Time Errors

Tapestry has some rich support for detecting and presenting errors at runtime.

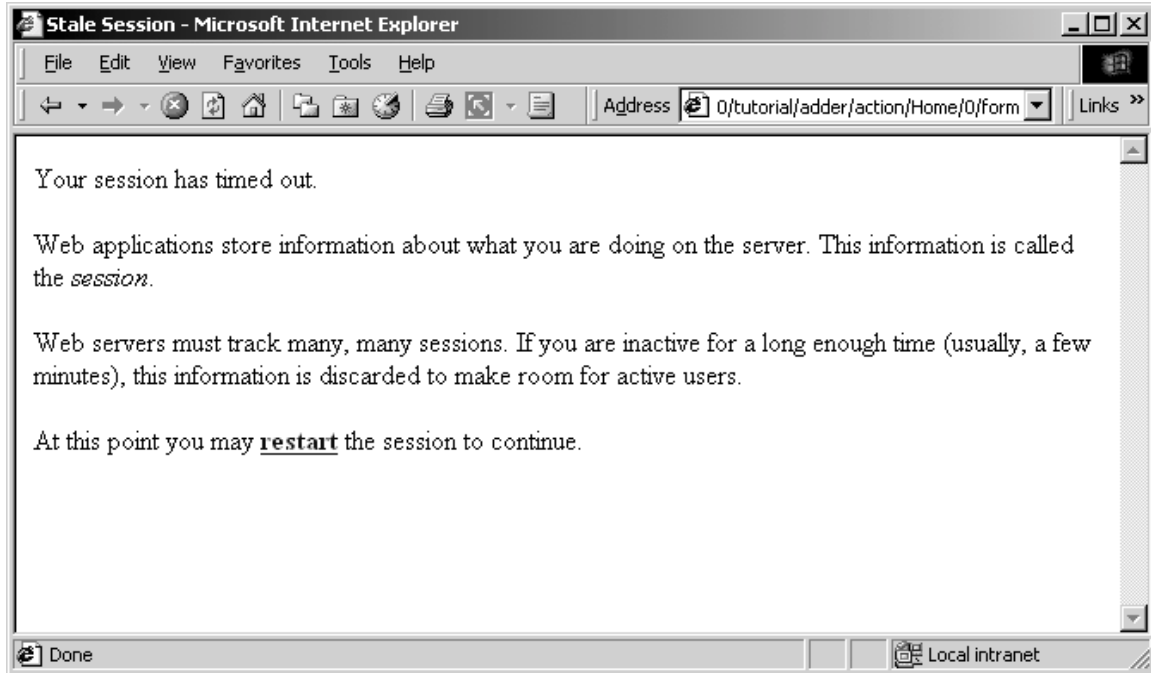
One of the benefits to developing using Tapestry is its robust exception handling support. We'll demonstrate these by creating invalid URLs.

Stale Sessions

As we just demonstrated, Tapestry is quite careful about conversational state. What happens if all the conversation state is lost?

Start up adder application then enter a few numbers. Go back to the window executing Jetty and stop it, then restart it.

Now, try to add an additional number to the list.



Because Tapestry can't find any information about your session, it assumes the session timed out and was discarded, and so presents the default error page for this situation.

Remember that most Tapestry URLs are very conversational, they only make sense as the most recent request in a series of requests exchanged between the client and the server.

This means that most pages in a Tapestry application can't be bookmarked; the URL that would be stored in the client's web browser is not meaningful. Creating bookmarkable pages is a subject of a later tutorial.

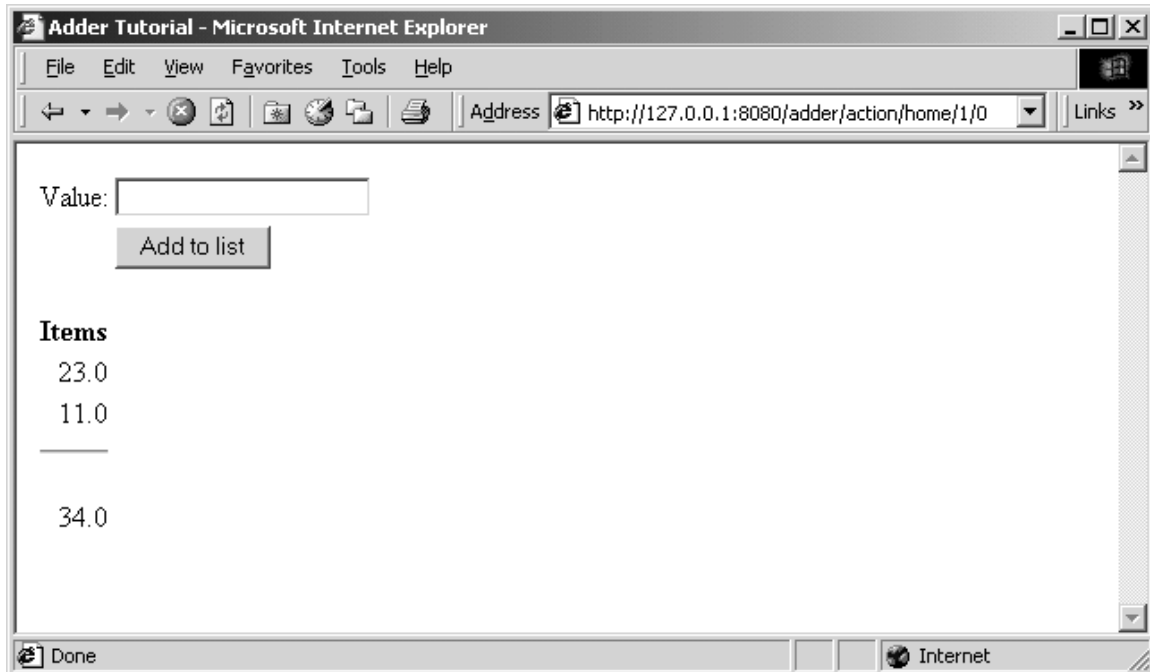
Exception Handling

Tapestry handles exceptions, catching them when they occur and formatting a readable page with all the details. Of course, in your own application, such exceptions will never occur, or will be caught and handled by your own code.

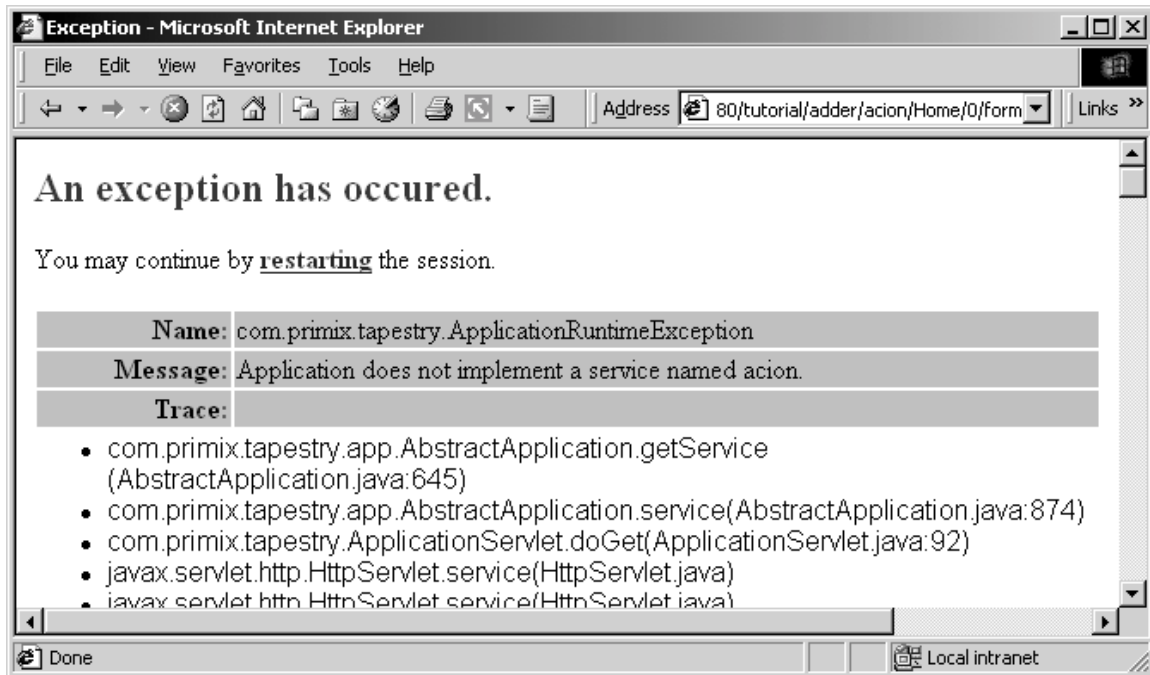
Still, it's nice that Tapestry can assist when debugging during development, when uncaught exceptions may in fact be thrown.

To demonstrate what Tapestry does for exceptions, we need to do a little bit of sneaky work.

First, enter a few numbers into the Adder application:



Now, edit the URL in the Address field, and change the word "action" to "acion" (i.e, remove the letter 't') and hit return.



Tapestry has discovered that the URL was invalid ... in this case that the word "action" was changed to "acion". Since Tapestry normally produces all the URLs it must later consume, it doesn't make an effort to pretty this up (as it does with stale links and sessions), instead it throws an exception which is caught and displayed.

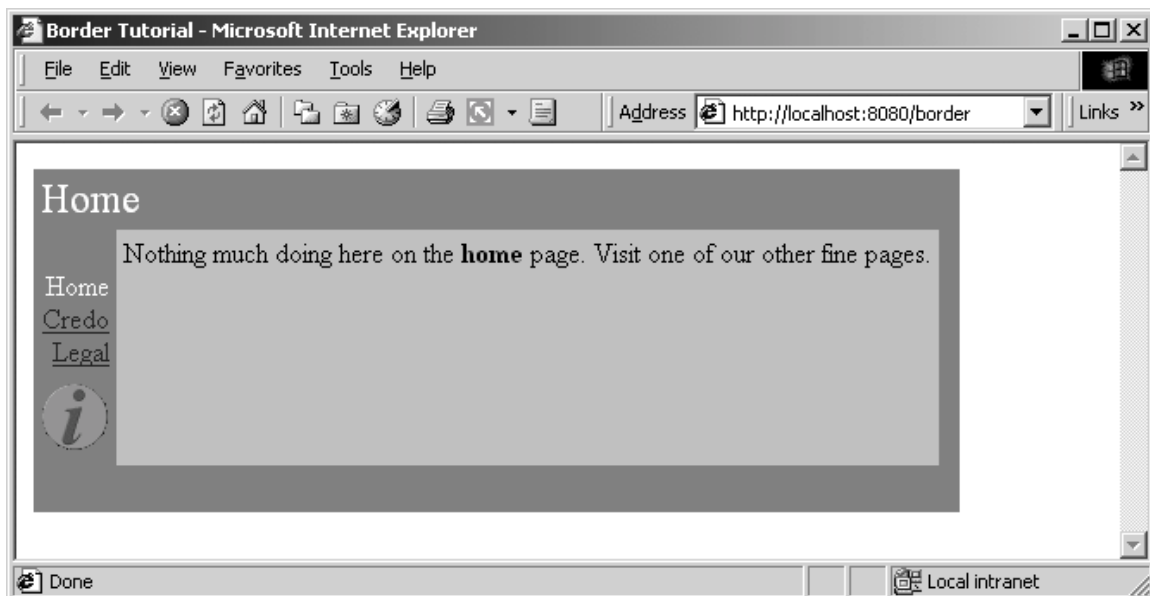
As you may notice, the exception report is extremely complex. Tapestry displays all the information it can about the exception that was thrown ... it can even break apart nested exceptions and dig down to the deepest one. It shows the stack trace where the deepest exception was thrown. It also provides information about the `HttpRequest`, `HttpSession`, `ServletContext` and Java VM.

Finally, it includes a link that will destroy the current `HttpSession` and restart the application from scratch.

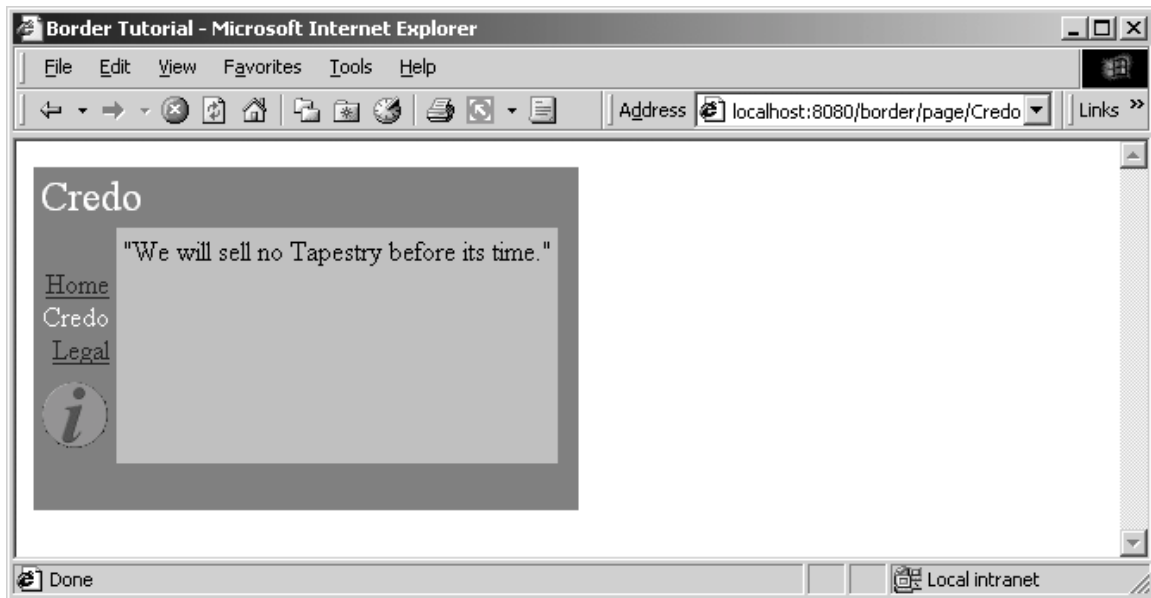
Re-usable Components

Tapestry is designed to facilitate the creation of re-usable components; this chapter will show an example of such a component.

In this tutorial, we'll show how to create a re-usable component. One common use of components is 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.



Navigating to another page results in a similar display:



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.

The "i" in the circle 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 separately, in the `tutorial.border` package.

Each of the three pages has a similar HTML template:

```

Home.html
<jwc id="border">

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

</jwc>

```

What we're doing here is wrapping the entire page inside the border. Note that we don't specify any `<HTML>` or `<BODY>` tags; those are provided by the border (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) get a chance to render. On the Home page, the Border component 'bats' first *and* cleanup.

The construction of the Border component is based on 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.

Border.html

```
<jwc id="shell">
<jwc id="body">
<table border=0 bgcolor=gray 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>
<jwc id="e">
  <br><jwc id="link"><jwc id="insertName"/></jwc>
</jwc>
  </font>
</td>
  <td rowspan=2 valign=top bgcolor=silver>
    <jwc id="wrapped"/>
  </td>
  <td rowspan=2 width=4></td>
</tr>
<tr>
  <td><jwc id="showInspector"/></td>
</tr>
<tr>
  <td colspan=3 height=4>&nbsp;</td>
</tr>
</table>
</jwc>
</jwc>
```

The `insertApplicationName` and `insertPageTitle` components provides the name of the application, and the title of the page within the application.

The `e`, `link` and `insertName` components provide the inter-page navigation links.

The `showInspector` component provides the button below the page names (the italicized "i" in a circle) and will be explained shortly.

The `shell` component provides the outermost portions of the page, the `<html>` and `<head>` tags.

The `body` component is a replacement for the `<body>` tag; it is required to support automatic rollover buttons (such as the `showInspector`) and will be used by most Tapestry applications.

Lastly, the `wrapped` component provides the actual content for the page.

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 border component. In fact, the application object provides the list.

Border.jwc

```
<?xml version="1.0"?>
<!DOCTYPE specification PUBLIC "-//Primix Solutions//Tapestry Specification
1.0//EN"
    "http://tapestry.sourceforge.net/dtd/Tapestry_1_0.dtd">
<specification>
  <class>tutorial.border.Border</class>
  <parameters>
    <allow-informal-parameters>no</allow-informal-parameters>
    <parameter>
      <name>title</name>
      <java-type>java.lang.String</java-type>
      <required>yes</required>
    </parameter>
    <parameter>
      <name>pages</name>
      <required>yes</required>
    </parameter>
  </parameters>
  <components>
    <component>
      <id>shell</id>
      <type>Shell</type>
      <bindings>
        <binding>
          <name>title</name>
          <property-
path>page.application.specification.name</property-path>
          </binding>
        </bindings>
      </component>
    <component>
      <id>insertPageTitle</id>
      <type>Insert</type>
      <bindings>
        <inherited-binding>
          <name>value</name>
          <parameter-name>title</parameter-name>
        </inherited-binding>
      </bindings>
    </component>
    <component>
      <id>body</id>
      <type>Body</type>
    </component>
    <component>
      <id>e</id>
      <type>Foreach</type>
      <bindings>
        <inherited-binding>
          <name>source</name>
          <parameter-name>pages</parameter-name>
```

```

        </inherited-binding>
        <binding>
            <name>value</name>
            <property-path>pageName</property-path>
        </binding>
    </bindings>
</component>
<component>
    <id>link</id>
    <type>Page</type>
    <bindings>
        <binding>
            <name>page</name>
            <property-path>pageName</property-path>
        </binding>
        <binding>
            <name>enabled</name>
            <property-path>enablePageLink</property-
path>
        </binding>
    </bindings>
</component>
<component>
    <id>insertName</id>
    <type>Insert</type>
    <bindings>
        <binding>
            <name>value</name>
            <property-path>pageName</property-path>
        </binding>
    </bindings>
</component>
<component>
    <id>wrapped</id>
    <type>InsertWrapped</type>
</component>
<component>
    <id>showInspector</id>
    <type>ShowInspector</type>
</component>
</components>
</specification>

```

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

We start by declaring two parameters: title and pages. The first is the title that will appear on the page. The second is the list of page names for the navigation area. We don't specify a type for pages because we want to allow all the possibilities (List, Iterator, Java array) that are acceptable as the source parameter to a Foreach.

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.

jumps from the page, to the application, to the application specification and gets the name of the application.

Further down we see that the `insertPageTitle` component inherits the title parameter from its container, the border component. Whatever binding is provided for the title parameter of the border will also be used as the value parameter of the `insertPageTitle` component. Using these inherited bindings simplifies the process of creating complex components from simple ones.

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 successive 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).

The link component creates the link to the other pages. It has an `enabled` parameter; when false the link component doesn't 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, `getEnablePageLink()`, that returns true unless the `pageName` parameter (set by the `e` component) matches the current page's name.

The `showInspector` component creates a rollover button (the "i" lights up when the mouse is moved over it):



Clicking on the button raises a second window that describes the current page in the application (this is used when debugging a Tapestry application). 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).

```

Home.jwc
<?xml version="1.0"?>
<!DOCTYPE specification PUBLIC

```

```

"-//Primitiv Solutions//Tapestry Specification 1.0//EN"
"http://tapestry.sourceforge.net/dtd/Tapestry_1_0.dtd">

<specification>
  <class>com.primix.tapestry.BasePage</class>

  <components>
    <component>
      <id>border</id>
      <type>Border</type>

      <bindings>
        <static-binding>
          <name>title</name>
          <value>Home</value>
        </static-binding>

        <binding>
          <name>pages</name>
          <property-path>application.pageNames</property-path>
        </binding>
      </bindings>
    </component>
  </components>

</specification>

```

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

BorderApplication.java (excerpt)

```

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

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

```

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

Border.application (excerpt)

```

<component>
  <alias>Border</alias>
  <type>/tutorial/border/Border.jwc</type>
</component>

```

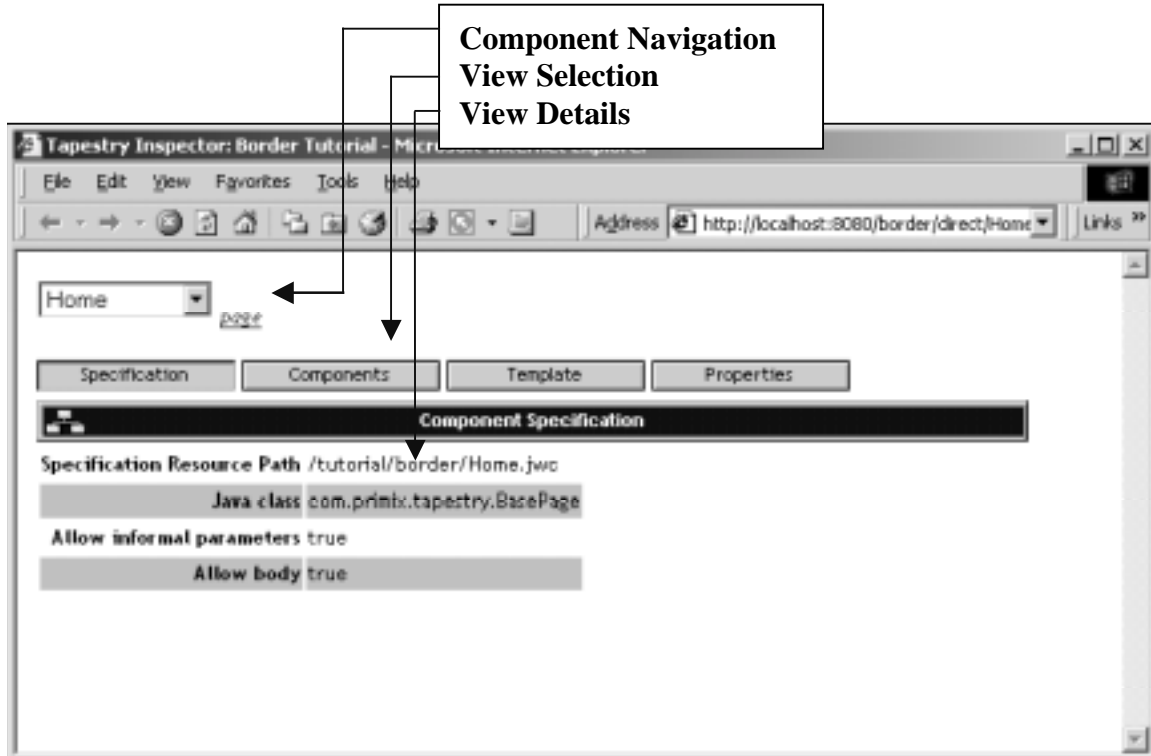
Had we failed to do this, we would have had to specify the complete resource path, `/tutorial/border/Border.jwc`, on each page's specification, instead of the short alias 'Border'. There is no magic about the existing Tapestry component types (Insert, ForEach, Page, etc.) ... they each have an alias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

The Tapestry Inspector

*Tapestry includes a powerful tool for debugging a running application, called *The Inspector*. This chapter will provide a brief overview of its abilities.*

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 circle with the italic "i"). A new window will launch, containing the 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.

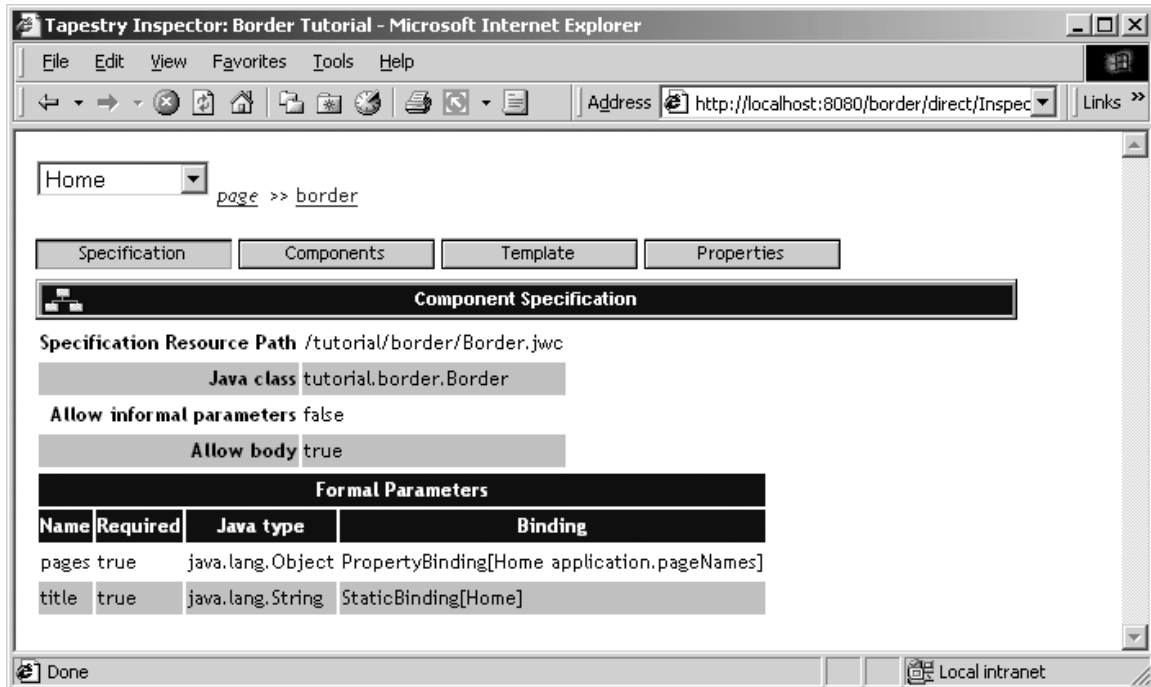
The Inspector consists of three sections. The Component Navigation allows the page to be selected (using the drop-down list), and then shows the page, or component on the page, being inspected.

The View Selection selects one of four views of the component to be selected. Finally, the View Details section shows detailed information about the component, as selected.

The ShowInspector component and the Inspector page are built-into the Tapestry framework. Any Tapestry application can make use of the Inspector by simply adding a ShowInspector component to any page. Most commonly, the ShowInspector component is added to a persistent navigation component, such as the Border component (in these examples).

Specification View

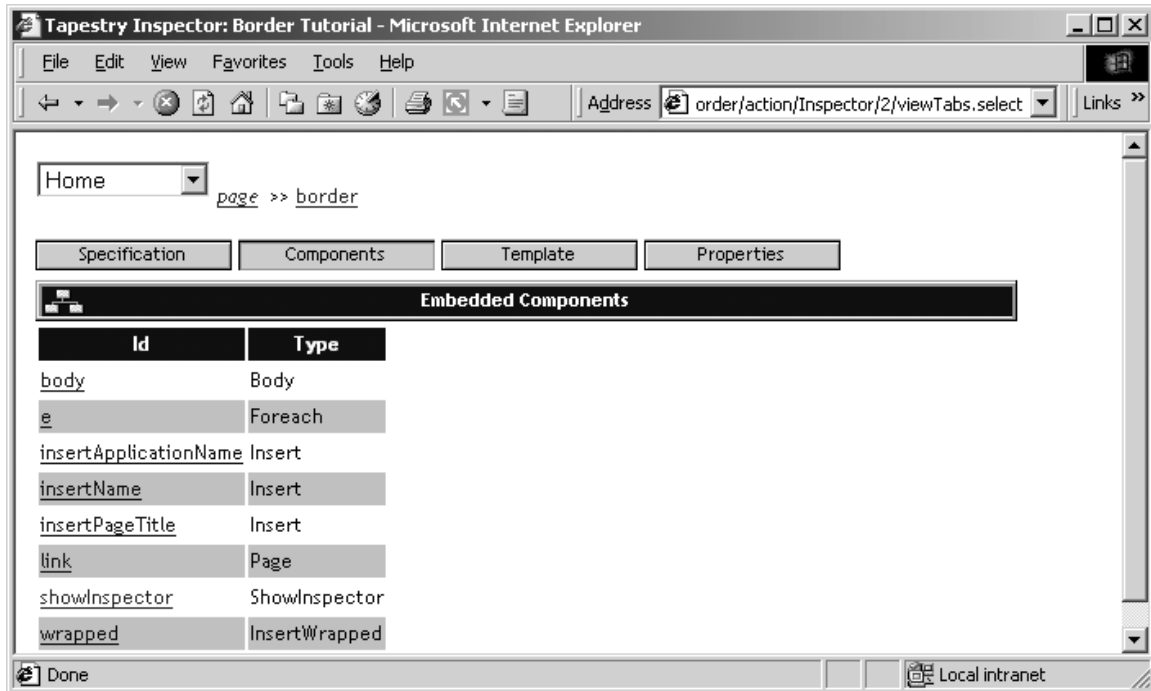
The specification tab show the basic information about the component, plus its formal and informal parameters (and how they are bound), and any assets.



A more complicated component may also show informal parameters and assets.

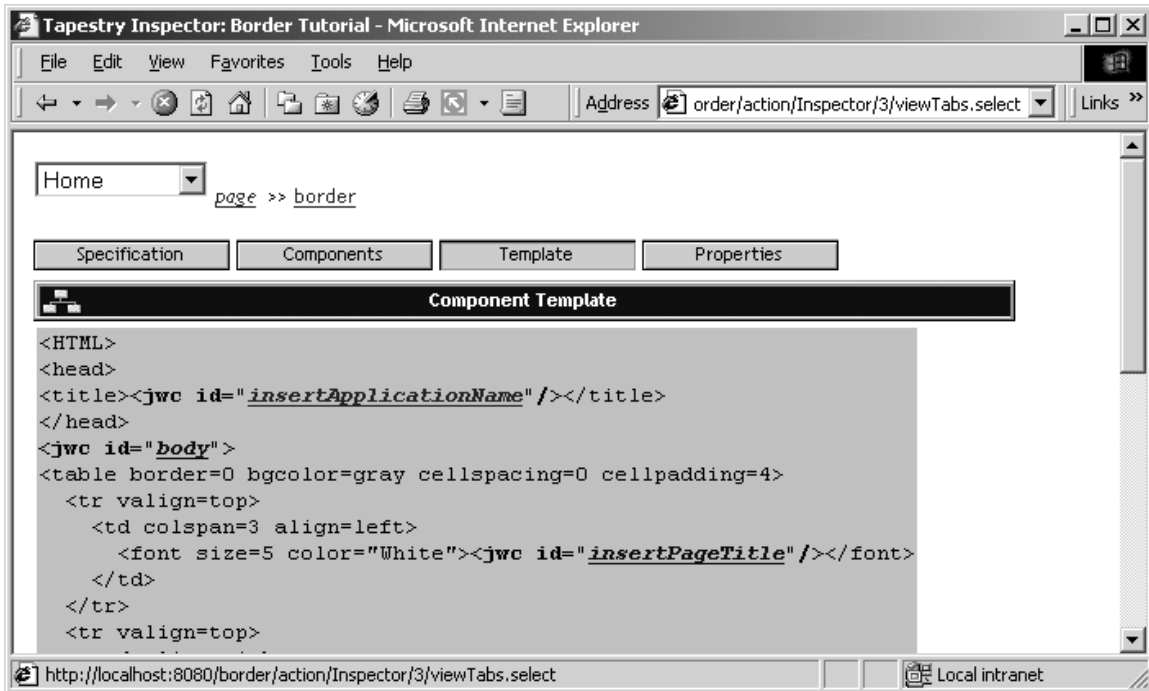
Components View

The components view provides a list of the components embedded in the currently inspected component. Clicking the name of a component inspects that component.



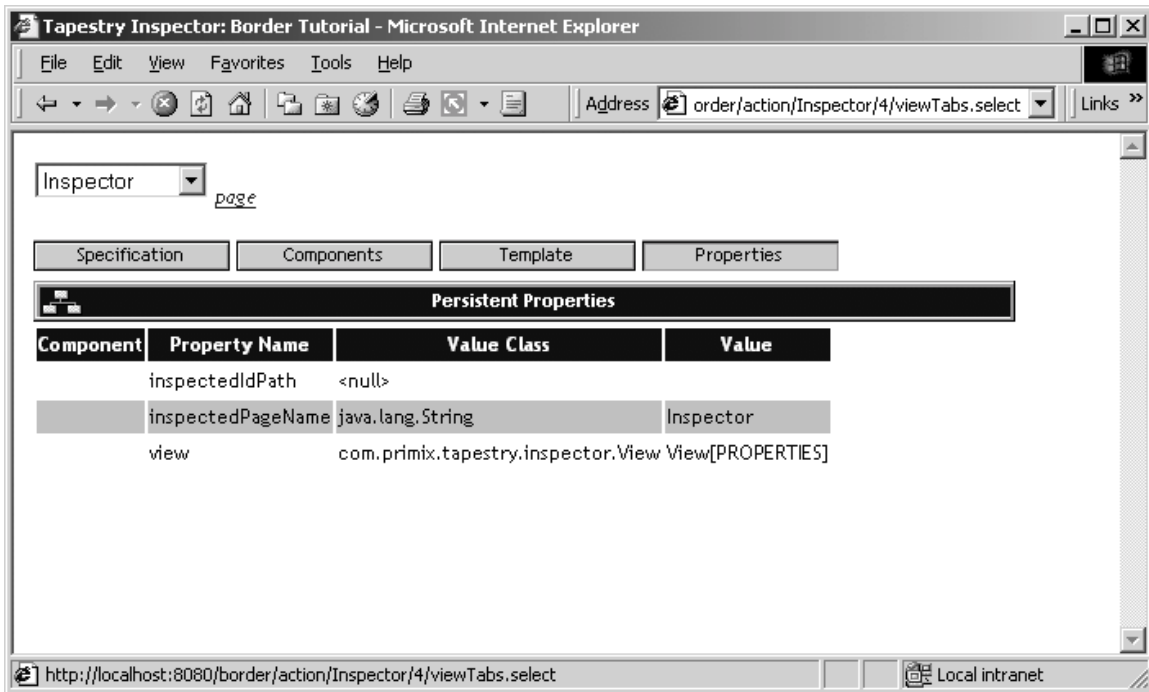
Template View

The template view shows the template (if known) for the component. It is formatted much like the examples in this document, with `<jwc>` tags in bold. In addition, component names are links, showing the template (if any) for the embedded component.



Properties View

The properties view shows all persistent properties for the page.



The component column is usually blank, except in the rare case that a component has its own persistent properties, in which case the component's id path is displayed.

Complex Forms and Output

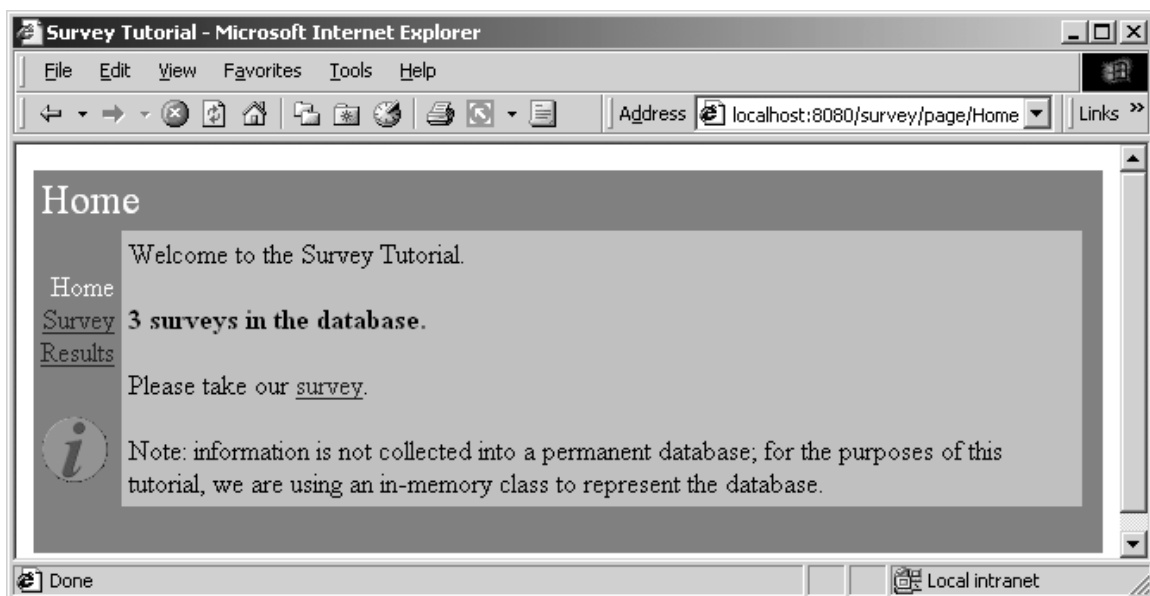
Demonstrates more of Tapestry's abilities to build and process form information, and produce complex output.

Tapestry includes a number of components designed to simplify interactions with the client, especially when handling forms.

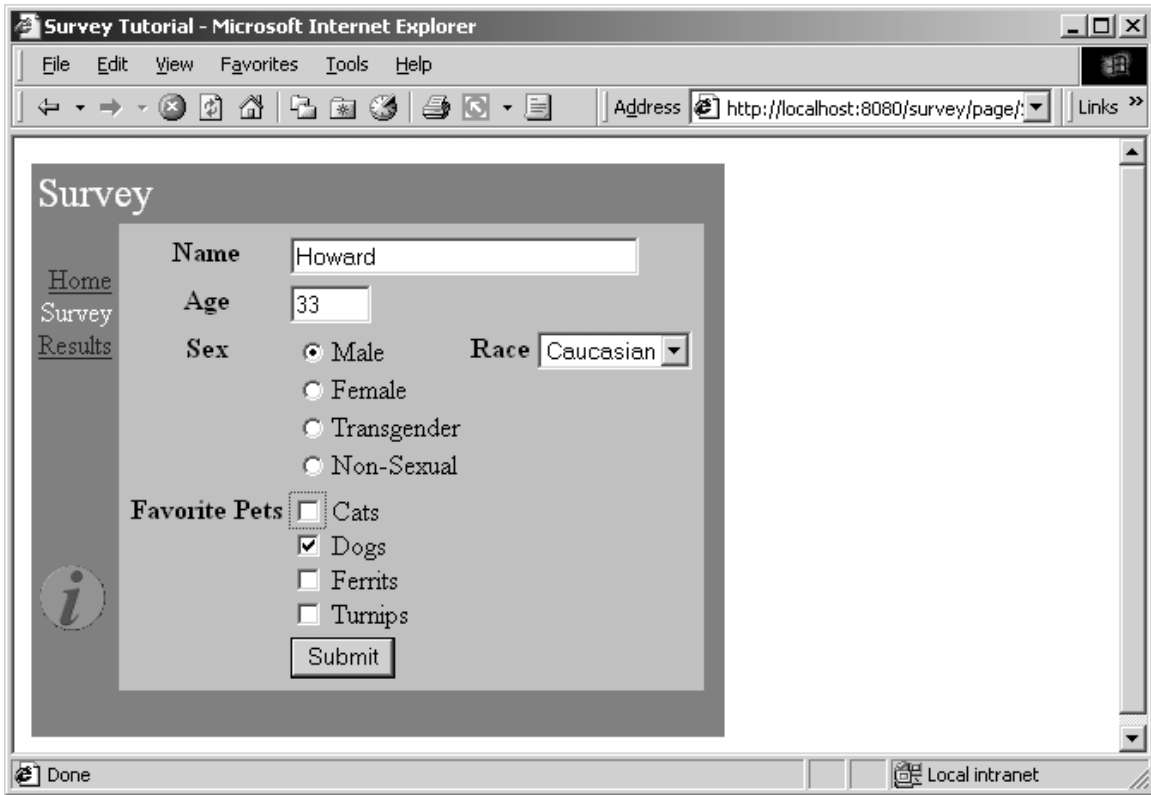
In this chapter, we'll build a survey-taking application that collects information from the user, stores it in an in-memory database, and produces tabular results summarizing what has been entered.

We'll see how to validate input from the client, how to create radio groups and pop-up selections and how to organize information for display.

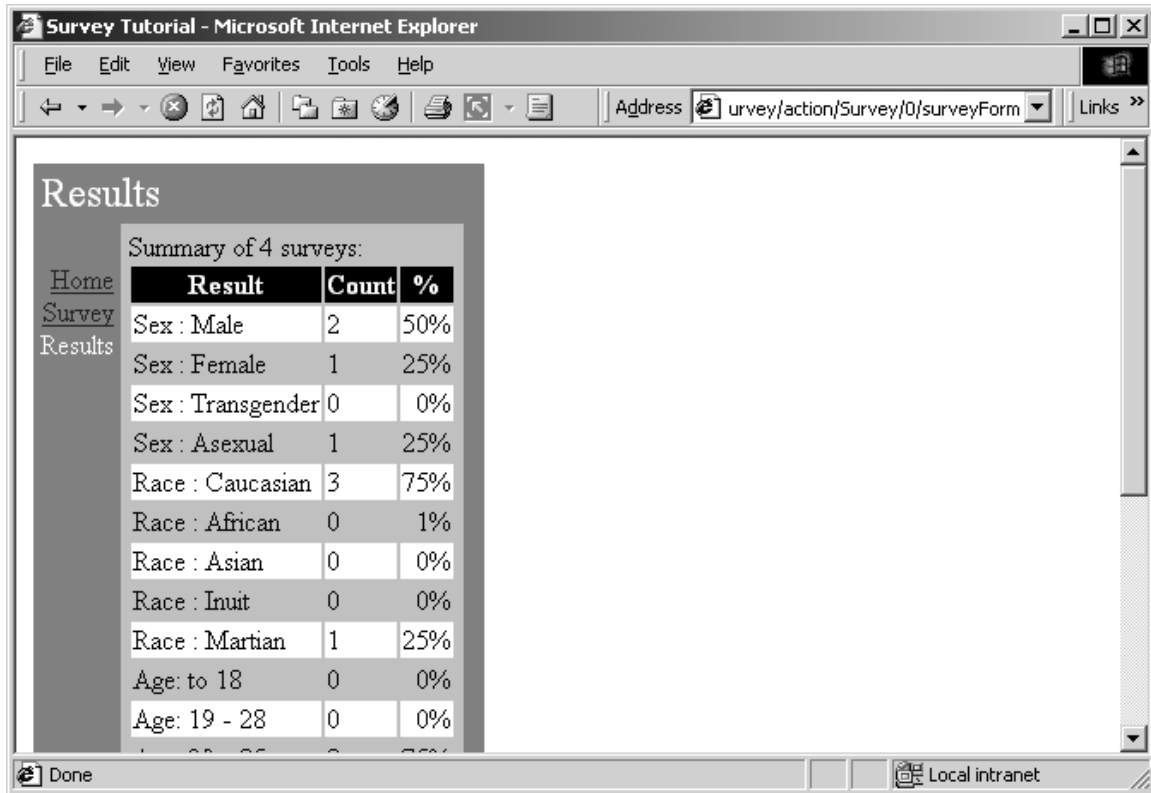
The application has three main screens; the first is a home page:



The second page is for entering survey data:



The last page is used to present results collected from many surveys:



In addition, we are re-using the Border component from the previous chapter.

The application does not use an actual database; the survey information is stored in memory (the amount of work to set up a JDBC database is beyond the scope of this tutorial).

The source code for this chapter is in the `tutorial.survey` package.

Survey

At the root of this application is an object that represents a survey taken by a user. We want to collect the name (which is optional), the sex and the race, the age and lastly, which pets the survey taker prefers.

Survey.java

```
package tutorial.survey;

import java.util.*;
import com.primix.tapestry.*;
import java.io.*;

public class Survey implements Serializable, Cloneable
{
    private Object primaryKey;
    private String name;
    private int age = 0;
    private Sex sex = Sex.MALE;
```

```
private Race race = Race.CAUCASIAN;

private boolean likesDogs = true;
private boolean likesCats;
private boolean likesFerrits;
private boolean likesTurnips;

public Object getPrimaryKey()
{
    return primaryKey;
}

public void setPrimaryKey(Object value)
{
    primaryKey = value;
}

public String getName()
{
    return name;
}

public void setName(String value)
{
    name = value;
}

public int getAge()
{
    return age;
}

public void setAge(int value)
{
    age = value;
}

public void setSex(Sex value)
{
    sex = value;
}

public Sex getSex()
{
    return sex;
}

public void setRace(Race value)
{
    race = value;
}

public Race getRace()
{
    return race;
}
```



```
public boolean getLikesCats()
{
    return likesCats;
}

public void setLikesCats(boolean value)
{
    likesCats = value;
}

public boolean getLikesDogs()
{
    return likesDogs;
}

public void setLikesDogs(boolean value)
{
    likesDogs = value;
}

public boolean getLikesFerrits()
{
    return likesFerrits;
}

public void setLikesFerrits(boolean value)
{
    likesFerrits = value;
}

public boolean getLikesTurnips()
{
    return likesTurnips;
}

public void setLikesTurnips(boolean value)
{
    likesTurnips = value;
}

/**
 * Validates that the survey is acceptable; throws an {@link
IllegalArgumentException}
 * if not valid.
 *
 */

public void validate()
throws IllegalArgumentException
{
    if (race == null)
        throw new IllegalArgumentException("Race must be
specified.");

    if (sex == null)
        throw new IllegalArgumentException("Sex must be
specified.");
```

```

        if (age < 1)
            throw new IllegalArgumentException("Age must be at least
one.");
    }

    public Object clone()
    {
        try
        {
            return super.clone();
        }
        catch (CloneNotSupportedException e)
        {
            return null;
        }
    }
}

```

The race and sex properties are defined in terms of the `Race` and `Sex` classes, which are derived from `com.primix.foundation.Enum`. Enum classes act like C enum types; a specific number of pre-defined values are declared by the class (as static final constants of the class).

Race.java

```

package tutorial.survey;

import com.primix.foundation.Enum;

/**
 * An enumeration of different races.
 */
public class Race extends Enum
{
    public static final Race CAUCASIAN = new Race("CAUCASIAN");
    public static final Race AFRICAN = new Race("AFRICAN");
    public static final Race ASIAN = new Race("ASIAN");
    public static final Race INUIT = new Race("INUIT");
    public static final Race MARTIAN = new Race("MARTIAN");

    private Race(String enumerationId)
    {
        super(enumerationId);
    }

    private Object readResolve()
    {
        return getSingleton();
    }
}

```

This is better than using `String` or `int` constants because of type safety; the Java compiler will

notice if you pass `Race.INUIT` as a parameter that expects an instance of `Sex` ... if they were both encoded as numbers, the compiler wouldn't know that there was a programming error.

SurveyDatabase

The `SurveyDatabase` class is a mockup of a database for storing Surveys, it has methods such as `addSurvey()` and `getAllSurveys()`. To emulate a database, it even allocates primary keys for surveys. Additionally, when surveys are added to the database, they are copied and when surveys are retrieved from the database, they are copied (that is, modifying a `Survey` instance after adding it to, or retrieving it from, the database doesn't affect the persistently stored Surveys within the database ... just as if they were in external storage).

SurveyApplication

The database is accessed via the `SurveyApplication` object.

```

SurveyApplication.java (excerpt)
private transient SurveyDatabase database;

public SurveyDatabase getDatabase()
{
    return database;
}

protected void setupForRequest(RequestContext context)
{
    super.setupForRequest(context);

    if (database == null)
    {
        String name = "Survey.database";
        ServletContext servletContext;

        servletContext =
context.getServlet().getServletContext();

        database =
(SurveyDatabase)servletContext.getAttribute(name);

        if (database == null)
        {
            database = new SurveyDatabase();
            servletContext.setAttribute(name, database);
        }
    }
}

```

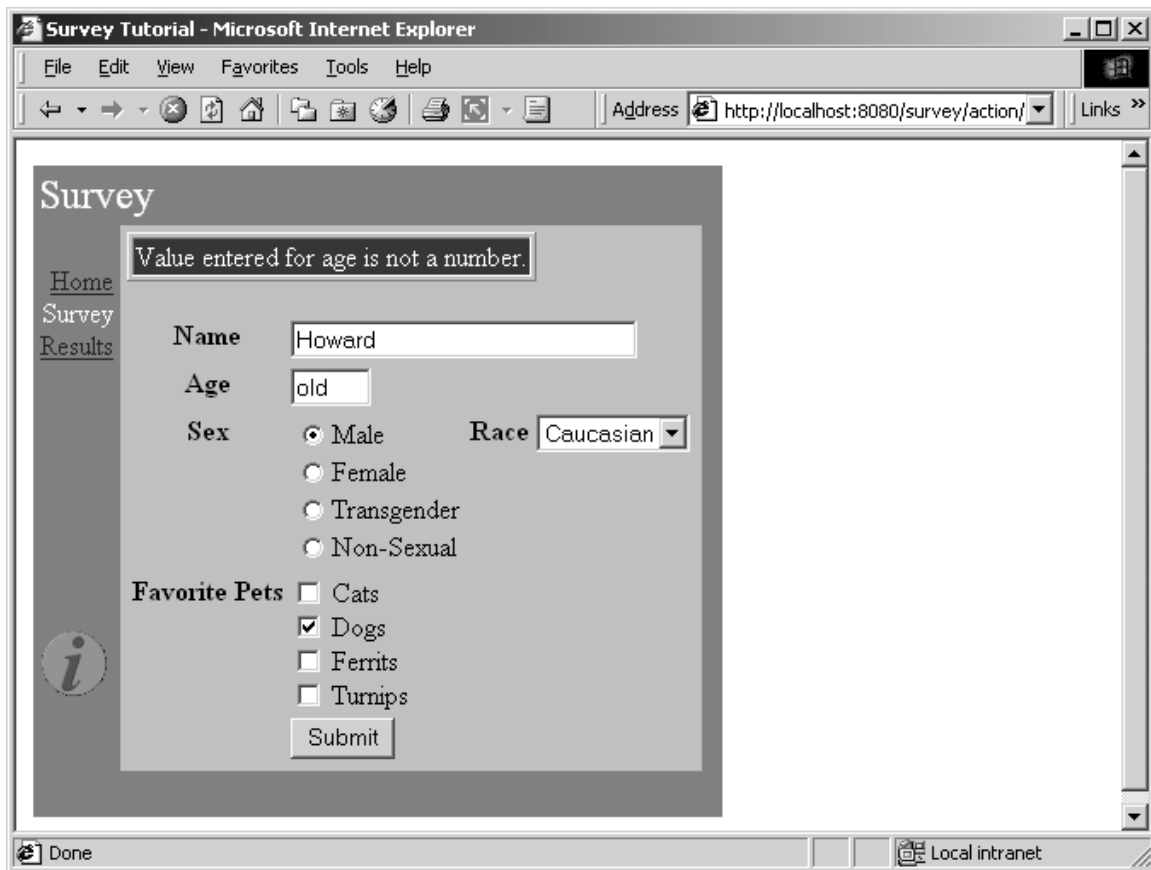
The `SurveyDatabase` instance is stored as a named attribute of the `ServletContext`, a shared space available to all sessions.

SurveyPage

The SurveyPage is where survey information is collected. It initially creates an Survey instance as a persistent page property. It uses Form and a number of other components to edit the survey.

When the survey is complete and valid, it is added to the database and the results page is used as an acknowledgment.

The SurveyPage also demonstrates how to validate data from a TextField component, and how to display validation errors. If invalid data is entered, then the user is notified (after submitting the form):



The HTML template for the page is relatively short. All the interesting stuff comes later, in the specification and the Java class.

SurveyPage.html

```
<jwc id="border">
<jwc id="ifError">
<table border=1>
<tr>
<td bgcolor=red>
<font style=bold color=white>
<jwc id="insertError"/>
```

```

</font> </tr> </tr> </table>
</jwc>

<jwc id="surveyForm">

<table border=0>

<tr valign=top> <th>Name</th>
    <td colspan=3><jwc id="inputName"/></td></tr>

<tr valign=top> <th>Age</th>
    <td colspan=3><jwc id="inputAge"/></td></tr>

<tr valign=top> <th>Sex</th>
    <td> <jwc id="inputSex"/>
    </td>

    <th>Race</th>

    <td><jwc id="inputRace"/>
    </td> </tr>

<tr valign=top> <th>Favorite Pets</th>
    <td colspan=3>
        <jwc id="inputCats"/> Cats
    <br><jwc id="inputDogs"/> Dogs
    <br><jwc id="inputFerrits"/> Ferrits
    <br><jwc id="inputTurnips"/> Turnips</td> </tr>
<tr>
<td></td>
<td colspan=3><input type=submit value="Submit"></td> </tr>
</table>

</jwc>
</jwc>

```

Most of this page is wrapped by the `surveyForm` component which is of type `Form`. The form contains two text fields (`nameField` and `ageField`), a group of radio buttons (`ageSelect`) and a pop-up list (`raceSelect`), and a number of check boxes (`cats`, `dogs`, `ferrits` and `turnips`).

Most of these components are pretty straight forward: `nameField` and `ageField` are setting `String` properties, and the check boxes are setting boolean properties. The two other components, `raceSelect` and `ageSelect`, are more interesting.

Both of these are of type `PropertySelection`; they are used for setting a specific property of some object to one of a number of possible values.

The `PropertySelection` component has some difficult tasks: It must know what the possible values are (including the correct order). It must also know how to display the values (that is, what labels to use on the radio buttons or in the pop up).

This information is provided by a model (an object that implement `com.primix.tapestry.components.IPropertySelectionModel`), an object that exists just to provide this information to a `PropertySelection` component.

There's a secondary question with `PropertySelection`: how the component is rendered. By default, it creates a pop-up list but this can be changed by providing an alternate renderer (using the component's `renderer` parameter). In our case, we used a secondary, radio-button renderer. Applications can also create their own renders, if they need to do something special with fonts, styles or images.

First, let's review the specification for the `SurveyPage`:

SurveyPage.jwc

```
<?xml version="1.0"?>
<!DOCTYPE specification PUBLIC "-//Primix Solutions//Tapestry Specification
1.0//EN"
    "http://tapestry.sourceforge.net/dtd/Tapestry_1_0.dtd">
<specification>
  <class>tutorial.survey.SurveyPage</class>
  <components>
    <component>
      <id>border</id>
      <type>Border</type>
      <bindings>
        <static-binding>
          <name>title</name>
          <value>Survey</value>
        </static-binding>
        <binding>
          <name>pages</name>
          <property-
path>application.pageNames</property-path>
        </binding>
      </bindings>
    </component>
    <component>
      <id>ifError</id>
      <type>Conditional</type>
      <bindings>
        <binding>
          <name>condition</name>
          <property-path>error</property-path>
        </binding>
      </bindings>
    </component>
    <component>
      <id>insertError</id>
      <type>Insert</type>
      <bindings>
        <binding>
          <name>value</name>
          <property-path>error</property-path>
        </binding>
      </bindings>
    </component>
    <component>
      <id>surveyForm</id>
      <type>Form</type>
      <bindings>
```

```

        <binding>
            <name>listener</name>
            <property-path>formListener</property-path>
        </binding>
    </bindings>
</component>
<component>
    <id>inputName</id>
    <type>TextField</type>
    <bindings>
        <static-binding>
            <name>displayWidth</name>
            <value>30</value>
        </static-binding>
        <static-binding>
            <name>maximumWidth</name>
            <value>100</value>
        </static-binding>
        <binding>
            <name>text</name>
            <property-path>survey.name</property-path>
        </binding>
    </bindings>
</component>
<component>
    <id>inputAge</id>
    <type>TextField</type>
    <bindings>
        <static-binding>
            <name>displayWidth</name>
            <value>4</value>
        </static-binding>
        <static-binding>
            <name>maximumWidth</name>
            <value>4</value>
        </static-binding>
        <binding>
            <name>text</name>
            <property-path>age</property-path>
        </binding>
    </bindings>
</component>
<component>
    <id>inputSex</id>
    <type>PropertySelection</type>
    <bindings>
        <binding>
            <name>value</name>
            <property-path>survey.sex</property-path>
        </binding>
        <binding>
            <name>model</name>
            <property-path>sexModel</property-path>
        </binding>
        <binding>
            <name>renderer</name>

```

```

        <property-
path>components.inputSex.defaultRadioRenderer</property-path>
        </binding>
    </bindings>
</component>
<component>
    <id>inputRace</id>
    <type>PropertySelection</type>
    <bindings>
        <binding>
            <name>value</name>
            <property-path>survey.race</property-path>
        </binding>
        <binding>
            <name>model</name>
            <property-path>raceModel</property-path>
        </binding>
    </bindings>
</component>
<component>
    <id>inputCats</id>
    <type>Checkbox</type>
    <bindings>
        <binding>
            <name>selected</name>
            <property-path>survey.likesCats</property-
path>
            </binding>
        </bindings>
    </component>
<component>
    <id>inputDogs</id>
    <type>Checkbox</type>
    <bindings>
        <binding>
            <name>selected</name>
            <property-path>survey.likesDogs</property-
path>
            </binding>
        </bindings>
    </component>
<component>
    <id>inputFerrits</id>
    <type>Checkbox</type>
    <bindings>
        <binding>
            <name>selected</name>
            <property-
path>survey.likesFerrits</property-path>
            </binding>
        </bindings>
    </component>
<component>
    <id>inputTurnips</id>
    <type>Checkbox</type>
    <bindings>
        <binding>

```



```

                <name>selected</name>
                <property-
path>survey.likesTurnips</property-path>
                </binding>
            </bindings>
        </component>
    </components>
</specification>

```

Several of the components, such as `inputName` and `inputTurnips`, modify properties of the survey directly. The `SurveyPage` class has a `survey` property, which allows for property paths like `survey.name` and `survey.likesTurnips`.

The age field is more complicated, since it must be converted from a `String` to an `int` before being assigned to the survey's age property ... and the page must check that the user entered a valid number as well.

Finally, the `SurveyPage` class shows how all the details fit together:

SurveyPage.java

```

package tutorial.survey;

import com.primix.tapestry.*;
import com.primix.tapestry.components.*;
import java.util.*;

public class SurveyPage extends BasePage
{
    private Survey survey;
    private String error;
    private String age;
    private IPropertySelectionModel sexModel;
    private IPropertySelectionModel raceModel;

    public IPropertySelectionModel getRaceModel()
    {
        if (raceModel == null)
            raceModel = new EnumPropertySelectionModel(
                new Race[]
                {
                    Race.CAUCASIAN, Race.AFRICAN, Race.ASIAN,
Race.INUIT, Race.MARTIAN
                }, getBundle("tutorial.survey.SurveyStrings"),
"Race");

        return raceModel;
    }

    public IPropertySelectionModel getSexModel()
    {
        if (sexModel == null)
            sexModel = new EnumPropertySelectionModel(
                new Sex[]
                {

```

```

        Sex.MALE, Sex.FEMALE, Sex.TRANSGENDER,
Sex.ASEXUAL
        }, getBundle("tutorial.survey.SurveyStrings"),
"Sex");

        return sexModel;
    }

    private ResourceBundle getBundle(String resourceName)
    {
        return ResourceBundle.getBundle(resourceName, getLocale());
    }

    public IActionListener getFormListener()
    {
        return new IActionListener()
        {
            public void actionTriggered(IComponent component,
IRequestCycle cycle)
            {
                try
                {
                    survey.setAge(Integer.parseInt(age));

                    survey.validate();
                }
                catch (NumberFormatException e)
                {
                    // NumberFormatException doesn't provide
any useful data

                    setError("Value entered for age is not a
number.");

                    return;
                }
                catch (Exception e)
                {
                    setError(e.getMessage());
                    return;
                }

                // Survey is OK, add it to the database.

                ((SurveyApplication)getApplication()).getDatabase().addSurvey(survey
);

                setSurvey(null);

                // Jump to the results page to show the totals.
                cycle.setPage("Results");
            }
        };
    }

    public Survey getSurvey()

```

```
{
    if (survey == null)
        setSurvey(new Survey());

    return survey;
}

public void setSurvey(Survey value)
{
    survey = value;
    fireObservedChange("survey", survey);
}

public void detachFromApplication()
{
    super.detachFromApplication();

    survey = null;
    error = null;
    age = null;

    // We keep the models, since they are stateless
}

public void setError(String value)
{
    error = value;
}

public String getError()
{
    return error;
}

public String getAge()
{
    int ageValue;

    if (age == null)
    {
        ageValue = getSurvey().getAge();

        if (ageValue == 0)
            age = "";
        else
            age = Integer.toString(ageValue);
    }

    return age;
}

public void setAge(String value)
{
    age = value;
}
}
```

A few notes. First, the `raceModel` and `sexModel` properties are created on-the-fly as needed. The `EnumPropertySelectionModel` is a provided class that simplifies using a `PropertySelection` component to set an `Enum`-typed property. We provide the list of possible values, and the information needed to extract the corresponding labels from a properties file, in this case, `SurveyStrings.properties`:

```

SurveyStrings.properties
Race.CAUCASIAN=Caucasian
Race.AFRICAN=African
Race.ASIAN=Asian
Race.INUIT=Inuit
Race.MARTIAN=Martian

Sex.ASEXUAL=Non-Sexual
Sex.MALE=Male
Sex.FEMALE=Female
Sex.TRANSGENDER=Transgender

```

Only `survey` is a persistent page property. The `error` property is transient (it is set to null at the end of the request cycle). The `error` property doesn't need to be persistent ... it is generated during a request cycle and is not used on a subsequent request cycle (because the survey will be re-validated).

Likewise, the `age` property isn't page persistent. If an invalid value is submitted, then its value will come up from the `HttpServletRequest` parameter and be plugged into the `age` property of the page. If validation of the survey fails, then the `SurveyPage` will be used to render the HTML response, and the invalid `age` value will still be there.

In the `detachFromApplication()` method, the `survey`, `error` and `age` properties are properly cleared. The `raceModel` and `ageModel` properties are not ... they are stateless and leaving them in place saves the trouble of creating identical objects later.

Results

Displaying results is broken up into two parts. In the first part, the database is queried for all surveys, and totals in a number of categories are prepared.

In the second part, those interim results are incorporated into the HTML response page.

```

Results.html
<jwc id="border">

Summary of <jwc id="insertSurveyCount"/> surveys:

    <jwc id="e-results">
        <jwc id="ifFirst">
<table border=0>
    <tr bgcolor=black>
        <th><font color=white>Result</font></th>

```

```

        <th><font color=white>Count</font></th>
        <th><font color=white>%</font></th>
    </tr>
    </jwc>
    <jwc id="results-row">
        <td>
            <jwc id="insertResult"/>
        </td>
        <td>
            <jwc id="insertCount"/>
        </td>
        <td align=right>
            <jwc id="insertPercent"/>
        </td>
    </jwc>
    <jwc id="ifLast">
</table>
    </jwc>
</jwc>

```

This template shows how those results will be provided to a Foreach component (e-results) that will iterate through them, and use a set of three Insert components. The ifFirst and ifLast components are used to generate the start and end of the HTML table (if the results are empty then the table doesn't get rendered at all).

The results-row component will take the place of the normal <TR> element in a table. It exists to vary the HTML bgcolor attribute, alternating between white and grey backgrounds for readability.

Results.jwc

```

<?xml version="1.0"?>
<!DOCTYPE specification PUBLIC
    "-//Primix Solutions//Tapestry Specification 1.0//EN"
    "http://tapestry.sourceforge.net/dtd/Tapestry_1_0.dtd">

<specification>
    <class>tutorial.survey.Results</class>

    <components>
        <component>
            <id>border</id>
            <type>Border</type>

            <bindings>
                <static-binding>
                    <name>title</name>
                    <value>Results</value>
                </static-binding>

                <binding>
                    <name>pages</name>
                    <property-
path>application.pageNames</property-path>

```

```

        </binding>
      </bindings>
    </component>

    <component>
      <id>insertSurveyCount</id>
      <type>Insert</type>

      <bindings>
        <binding>
          <name>value</name>
          <property-
path>database.surveyCount</property-path>
        </binding>
      </bindings>
    </component>

    <!-- The results is a List of Maps. -->

    <component>
      <id>e-results</id>
      <type>Foreach</type>

      <bindings>
        <binding>
          <name>source</name>
          <property-path>results</property-path>
        </binding>
      </bindings>
    </component>

    <component>
      <id>ifFirst</id>
      <type>Conditional</type>

      <bindings>
        <binding>
          <name>condition</name>
          <property-path>components.e-
results.first</property-path>
        </binding>
      </bindings>
    </component>

    <!-- This stands in for the TR element, but handles the
bgcolor. -->

    <component>
      <id>results-row</id>
      <type>Any</type>

      <bindings>
        <static-binding>
          <name>element</name>
          <value>tr</value>
        </static-binding>

```

```

        <binding>
            <name>bgcolor</name>
            <property-path>rowColor</property-path>
        </binding>
    </bindings>
</component>

<component>
    <id>insertResult</id>
    <type>Insert</type>

    <bindings>
        <binding>
            <name>value</name>
            <property-path>components.e-
results.value.name</property-path>
        </binding>
    </bindings>
</component>

<component>
    <id>insertCount</id>
    <type>Insert</type>

    <bindings>
        <binding>
            <name>value</name>
            <property-path>components.e-
results.value.count</property-path>
        </binding>
    </bindings>
</component>

<component>
    <id>insertPercent</id>
    <type>Insert</type>

    <bindings>
        <binding>
            <name>value</name>
            <property-path>components.e-
results.value.percent</property-path>
        </binding>
    </bindings>
</component>

<component>
    <id>ifLast</id>
    <type>Conditional</type>

    <bindings>
        <binding>
            <name>condition</name>
            <property-path>components.e-
results.last</property-path>
        </binding>
    </bindings>
</component>

```

```

        </bindings>
    </component>

</components>

</specification>

```

The presentation relies on the Java class providing a results property. This property is a `List` of `Maps`. Each `Map` has three keys: `name`, `count` and `percent`. The rest of the logic is simply to break apart this `List` into `Maps` (as property-path components.e-results.value), and to pull out the values for the three keys.

Creating this results property consumes the bulk of the class:

Results.java

```

package tutorial.survey;

import com.primix.tapestry.*;
import java.util.*;
import java.text.*;
import java.awt.Color;

public class Results extends BasePage
{
    private SurveyDatabase surveyDatabase;
    private boolean oddRow = false;
    private NumberFormat percentFormat;

    public SurveyDatabase getDatabase()
    {
        if (surveyDatabase == null)
        {
            SurveyApplication surveyApplication;

            surveyApplication = (SurveyApplication)application;

            surveyDatabase = surveyApplication.getDatabase();
        }

        return surveyDatabase;
    }

    public void detachFromApplication()
    {
        super.detachFromApplication();

        surveyDatabase = null;
        oddRow = false;
    }

    public String getRowColor()
    {
        Color color;
        String result;
    }
}

```



```

        if (oddRow)
            color = Color.lightGray;
        else
            color = Color.white;

        result = RequestContext.encodeColor(color);

        oddRow = !oddRow;

        return result;
    }

    public List getResults()
    {
        int raceAfrican = 0;
        int raceAsian = 0;
        int raceCaucasian = 0;
        int raceInuit = 0;
        int raceMartian = 0;
        int sexAsexual = 0;
        int sexFemale = 0;
        int sexMale = 0;
        int sexTransgender = 0;
        int likesCats = 0;
        int likesDogs = 0;
        int likesFerrits = 0;
        int likesTurnips = 0;
        int ageToTeen = 0; // 1 - 18
        int ageEarlyAdult = 0; // 19 - 28
        int ageToMiddle = 0; // 29 - 35
        int ageMiddle = 0; // 36 - 49
        int ageOlder = 0; // 50 - 64
        int ageRetire = 0; // 65 - 80
        int ageOld = 0; // 81 - 100
        Survey[] surveys;
        Survey survey;
        List result;
        Race race;
        Sex sex;
        int count;
        int i;
        int age;

        surveys = getDatabase().getAllSurveys();
        if (surveys == null ||
            surveys.length == 0)
            return null;

        count = surveys.length;
        for (i = 0; i < count; i++)
        {
            survey = surveys[i];

            race = survey.getRace();
            if (race == Race.AFRICAN)
                raceAfrican++;
        }
    }
}

```

```
if (race == Race.ASIAN)
    raceAsian++;

if (race == Race.CAUCASIAN)
    raceCaucasian++;

if (race == Race.INUIT)
    raceInuit++;

if (race == Race.MARTIAN)
    raceMartian++;

sex = survey.getSex();
if (sex == Sex.MALE)
    sexMale++;

if (sex == Sex.FEMALE)
    sexFemale++;

if (sex == Sex.TRANSGENDER)
    sexTransgender++;

if (sex == Sex.ASEXUAL)
    sexAsexual++;

if (survey.getLikesCats())
    likesCats++;

if (survey.getLikesDogs())
    likesDogs++;

if (survey.getLikesFerrits())
    likesFerrits++;

if (survey.getLikesTurnips())
    likesTurnips++;

age = survey.getAge();

if (age < 19)
    ageToTeen++;

if (age >= 19 && age <= 28)
    ageEarlyAdult++;

if (age >= 29 && age <= 35)
    ageToMiddle++;

if (age >= 36 && age <= 49)
    ageMiddle++;

if (age >= 50 && age <= 64)
    ageOlder++;

if (age >= 65 && age <= 80)
    ageRetire++;
```

```

        if (age >= 81)
            ageOld++;

    }

    result = new ArrayList();

    result.add(buildResult("Sex : Male", sexMale, count));
    result.add(buildResult("Sex : Female", sexFemale, count));
    result.add(buildResult("Sex : Transgender", sexTransgender,
count));
    result.add(buildResult("Sex : Asexual", sexAsexual, count));

    result.add(buildResult("Race : Caucasian", raceCaucasian,
count));
    result.add(buildResult("Race : African", raceAfrican, count));
    result.add(buildResult("Race : Asian", raceAsian, count));
    result.add(buildResult("Race : Inuit", raceInuit, count));
    result.add(buildResult("Race : Martian", raceMartian, count));

    result.add(buildResult("Age: to 18", ageToTeen, count));
    result.add(buildResult("Age: 19 - 28", ageEarlyAdult, count));
    result.add(buildResult("Age: 29 - 35", ageToMiddle, count));
    result.add(buildResult("Age: 36 - 49", ageMiddle, count));
    result.add(buildResult("Age: 50 - 64", ageOlder, count));
    result.add(buildResult("Age: 65 - 80", ageRetire, count));
    result.add(buildResult("Age: 80 and up", ageOld, count));

    result.add(buildResult("Likes cats", likesCats, count));
    result.add(buildResult("Likes dogs", likesDogs, count));
    result.add(buildResult("Likes ferrits", likesFerrits, count));
    result.add(buildResult("Likes turnips", likesTurnips, count));

    return result;
}

private Map buildResult(String name, int count, int total)
{
    Map result;

    result = new HashMap(3);
    result.put("name", name);
    result.put("count", new Integer(count));

    if (percentFormat == null)
        percentFormat =
NumberFormat.getPercentInstance(getLocale());

    result.put("percent", percentFormat.format((double)count /
(double)total));

    return result;
}
}

```



Further Study

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

- Separation 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 Primix 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 also shows how to manage EJB references and handles.