PRIMIX SOLUTIONS

Core Labs

Tapestry Tutorial

CORE LABS

Tapestry Tutorial

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

Table of Contents

Chapter 1

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 preperation 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 elminate 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 a 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.

These examples were developed using the Kawa IDE, JDK 1.2.2 and ServletExec Debugger 2.2.

Chapter

2

Setting up Kawa and ServletExec Debugger

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

First, create a working directory. In these examples, we'll use the working directory `D:\Work`. If this is not convienent, any directory may be chosen, but the reader will be responsible for adjusting the paths appropriately.

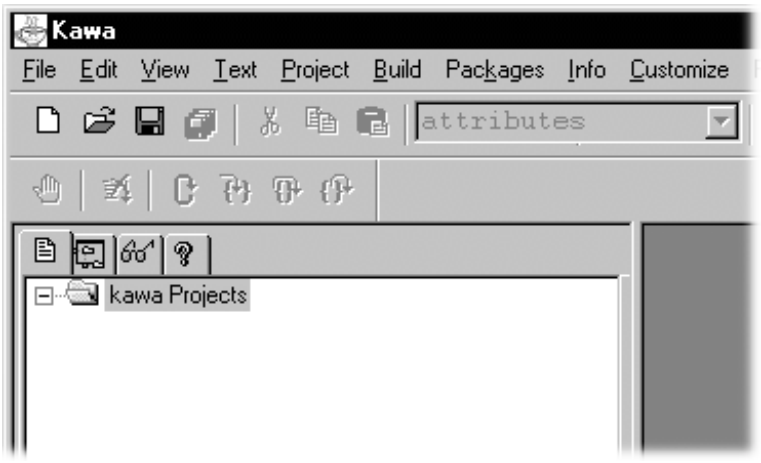
Directories within the working directory each store a Kawa project. In addition, the directory `D:\Work\lib` will contain copies of serveral Jar files needed when compiling and executing the tutorial.

Jar	**Description**
gnu-regexp.jar	GNU Regular Expression parser for Java, version 1.0.8. Used by the Tapestry framework.
jaxp.jar	Java API for XML Processing, version 1.0.1 (interface definitions).
parser.jar	Java API for XML Processing, version 1.0.1 (reference implementation).
ServletExecDebugger.jar	ServletExec Debugger, version 2.2. Includes the Java Servlet API version 2.1.1.
Tapestry.jar	Tapestry framework.

Extract the Tutorial files to `D:\Work\Tutorial`. You may have received the Tutorial files in a Zip file, or you may check them out of the CVS repository using WinCVS.

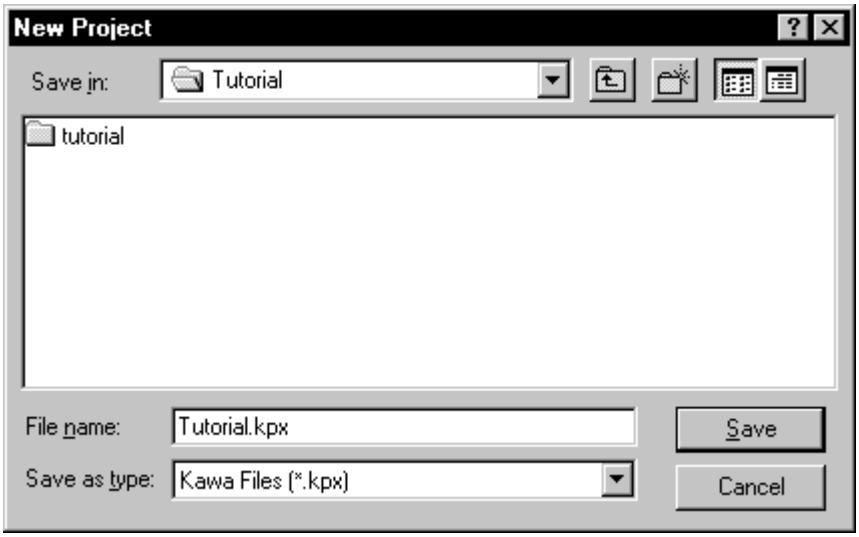
Setting up the Tutorial Project

Start with an empty Kawa workspace:



Now, create the Tutorial project by selecting Project ‣ New ... from the menu.

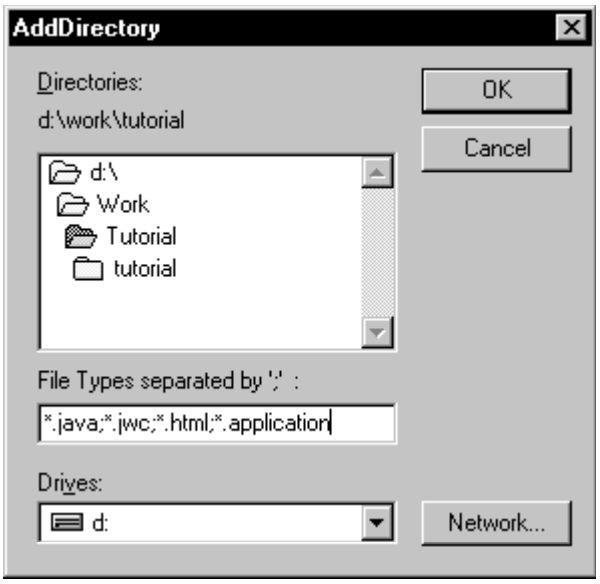
Navigate into the `D:\Work\Tutorial` directory, and enter `Tutorial.kpx` to create the new Project (you must specify the extension because of the naming conflict with the tutorial directory).



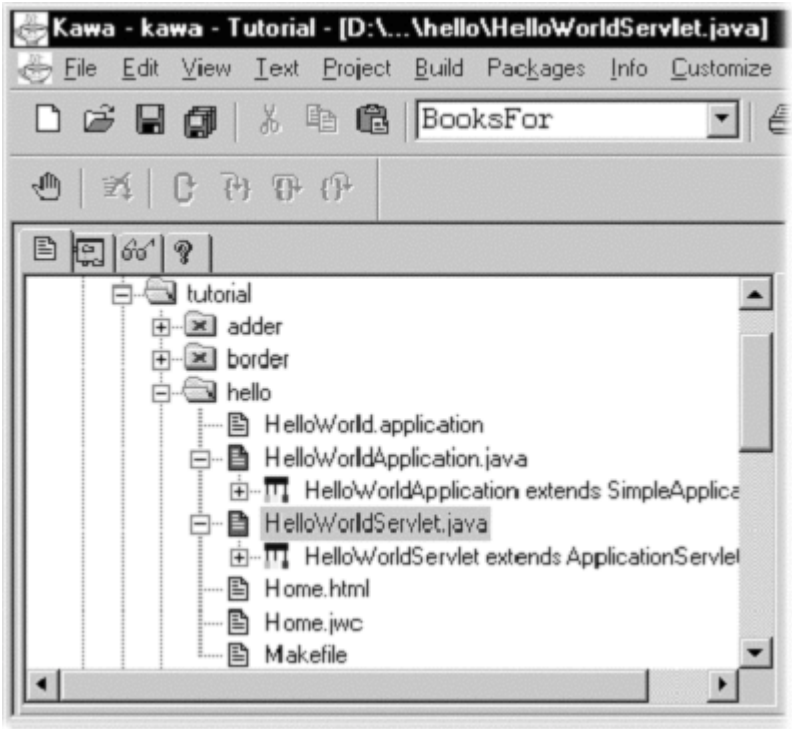
You now have a Kawa Project with no Java files. We'll now set up a list of files for the Tutorial project:

Select the Tutorial project, then select Project ‣ Add Directory ... from the menu.

We want to pick up not just the Java source files, but also Tapestry component specification files (.jwc), application specification (.application) and HTML templates (.html).



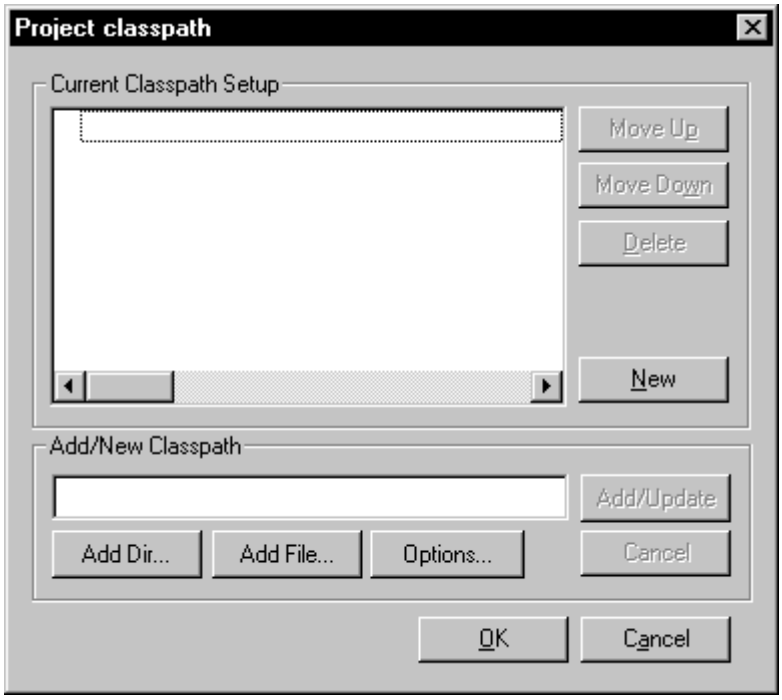
Kawa will locate these files and organize them into appropriate folders. You can then expand some folders to see the contents:



Now we have to setup the classpath for compiling this project.

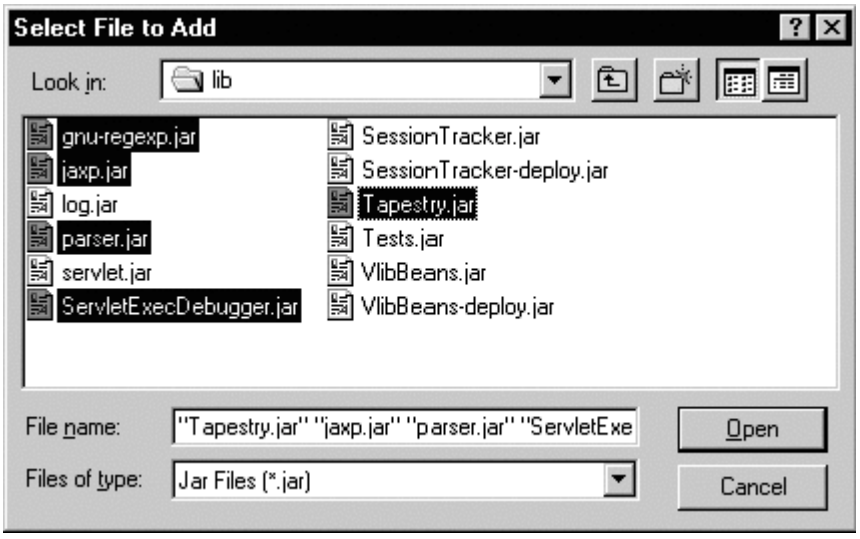
Select the Tutorial project and then choose Project ▸ Classpath ... from the menu.

Initially, the classpath is empty (except for classes provided by the JDK):

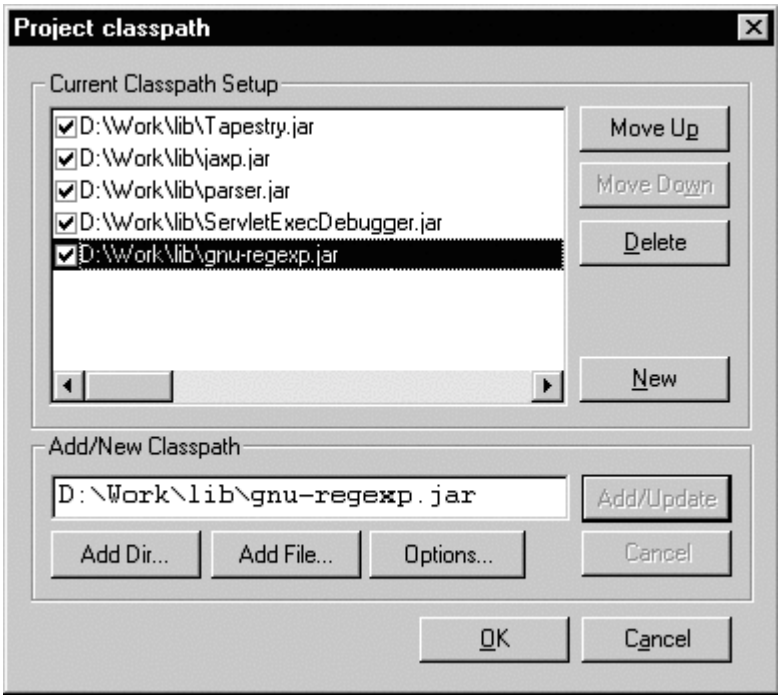


We need to add a few Jar files to the classpath. Click on the "Add File..." button towards the bottom of the panel:

You can select multiple files by holding down the control key while clicking:

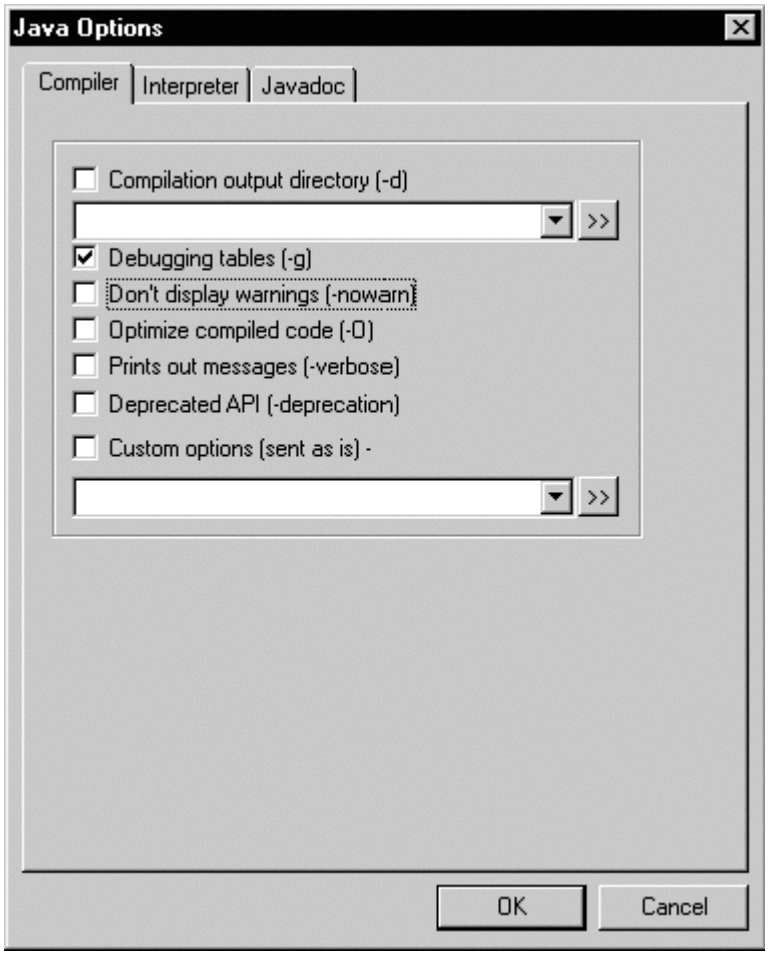


After returning to the previous panel, click "Update". The Jar files selected will be added to the classpath:



The order may be different, but that's irrelevant to Kawa and to the JDK. Kawa allows Jars to be easily added to or removed from the classpath using the checkboxes, but we want all of these Jars.

You can set many compilation options from within Kawa. Select the Tutorial project and choose Project ▸ Compiler Options... from the menu.



Turn on debugging output (for later, when we use Kawa to debug our application).

The project should now be compiled using the Project ‣ Rebuild All or Project ‣ Rebuild Dirty menu items.

ServletExec Debugger

ServletExec Debugger requires setting up two directories outside of the IDE. The first directory is where ServletExec Debugger stores configuration information about the different servlets. The second directory is the "web server" root directory (ServletExec Debugger acts as a simple web server, providing access to static resources such as GIF files as well as dynamic content from servlets).

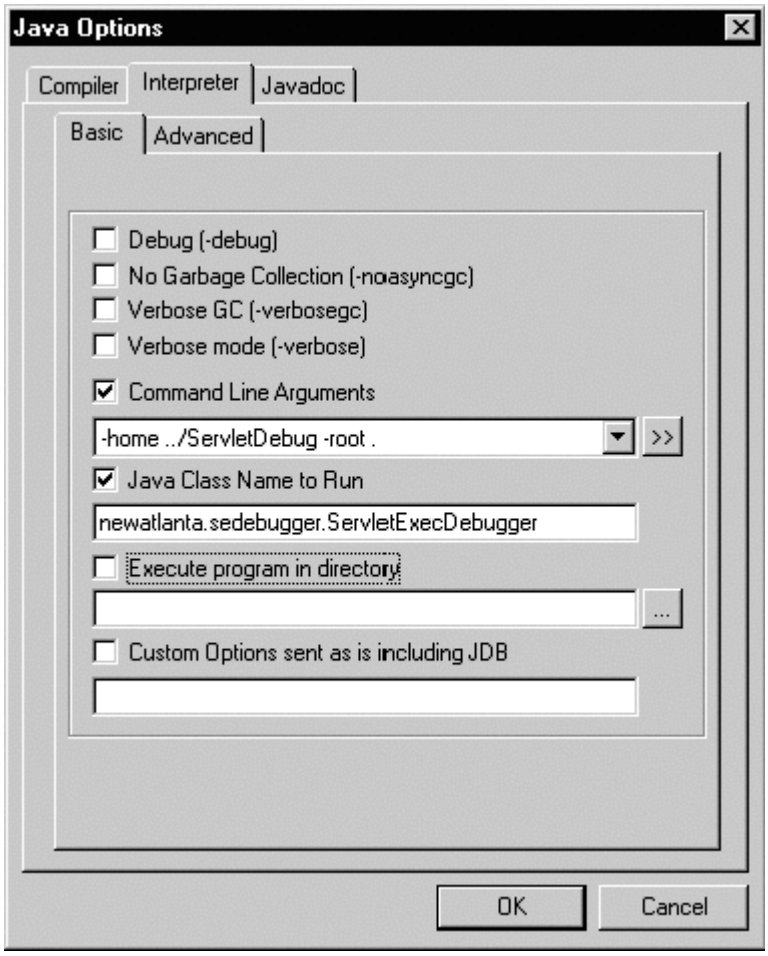
You don't need to create these directories first; ServletExec Debugger will create them the first time it starts up.

In my case, `D:\Work` was my main working directory, so I used `D:\Work\ServletDebug` for configuration, and the project directory as my web server root directory.

If you chose to put these files in a different directory, you'll have to adjust some of the examples in later chapters.

When we want to run or debug our applications, we don't execute a specific class for our application, we instead run the ServletExec Debugger, which acts as a simple web server and servlet container.

To set this up, you must select the Project ‣ Interpreter Options ... menu item, and update the command line arguments and Java class name.



Don't forget to click the checkboxes; as with the classpath, Kawa allows you to easily inlcude or exclude options used when running the program using those checkboxes; if they aren't checked, the option won't be included.

Chapter 3

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 demostrate 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., `D:\Work\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 a subclass the Tapestry class `com.primix.tapestry.app.SimpleApplication`.

`SimpleApplication` is an abstract class; we must provide implementations for two methods.

- `getSpecificationResourceName()` provides the resource path of the application's specification. The main purpose of the specification is to define the pages used in the application.

- `getSpecificationAttributeName()` provides a name used to store the specification in memory. The specification is needed to process every request; rather than parse it on each request, it is stored in memory for subsequent requests.

Tapestry can't use fixed names for either of these two values, since that would cause a naming conflict if two different Tapestry applications were run simultaneosly inside the same servlet container. This is not an unlikely possiblity ... a customer application will often be paired with a CSR (customer service representative) application.

In any case, the code for the application object is quite simple:

HelloWorldApplication.java

```
package tutorial.hello;

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

public class HelloWorldApplication extends SimpleApplication
{
  public HelloWorldApplication(RequestContext context, Locale locale)
  {
    super(context, locale);
  }

  protected String getSpecificationAttributeName()
  {
    return "Hello.specification";
  }

  protected String getSpecificationResourceName()
  {
    return "/tutorial/hello/HelloWorld.application";
  }
}
```

Application Servlet

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

This is all accomplished in a single method, `getApplication()`.

HelloWorldServlet.java

```
package tutorial.hello;

import com.primix.tapestry.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorldServlet extends ApplicationServlet
{
```

```
  protected IApplication getApplication(RequestContext context)
  {
    String name = "Hello.application";
    IApplication application;

    application = (IApplication)context.getSessionAttribute(name);

    if (application == null)
    {
      application = new HelloWorldApplication(context, null);
      context.setSessionAttribute(name, application);
    }

    return application;
  }
}
```

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 classpath. 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"?>
<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 a home page.

Home Page Specification

The home 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"?>

<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

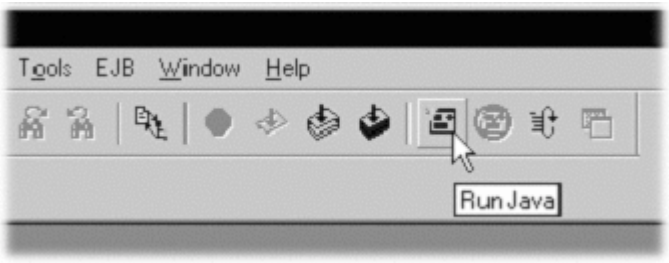
Finally, we get to the content of our application. This file is also a Java resource; it isn't directly visible to the web server. It has the same location and name as the component specification, except that it ends in "html".

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>
```

Launch ServletExec Debugger

The ServletExec Debugger server is launched using the "Run Java" toolbar button, by selecting Build ‣ Run from the menu, or by hitting **F4**.



Once launched. the output window will show the progress as the server starts up:

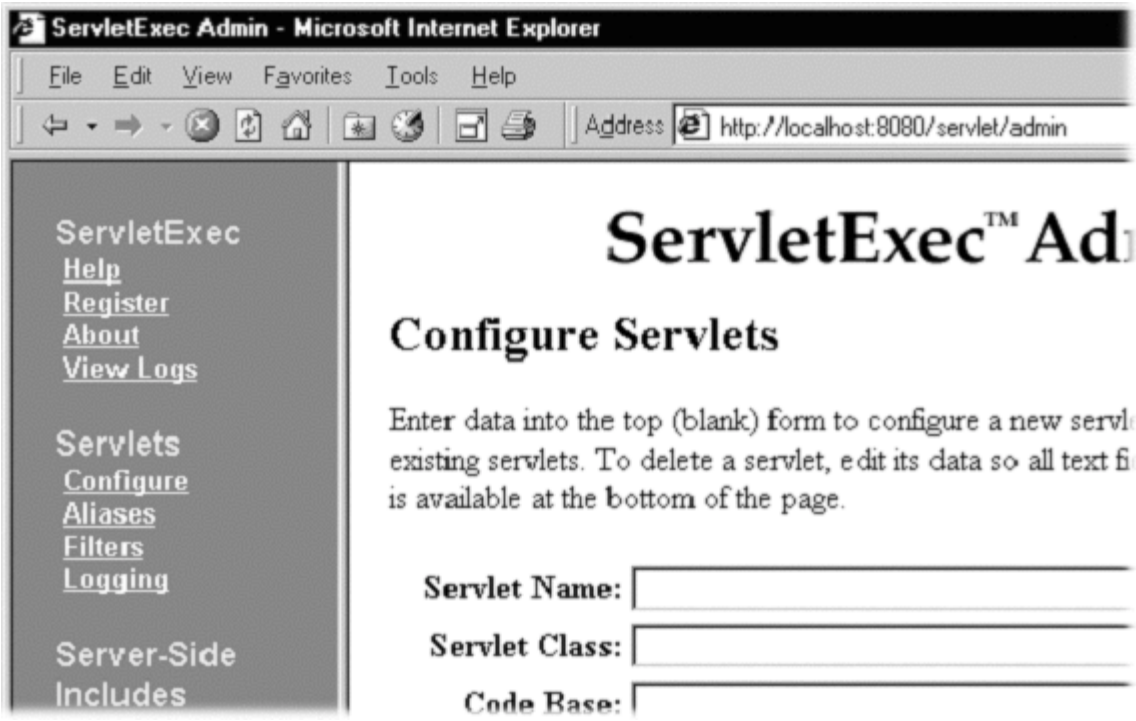
```
C:\jdk1.2.2\bin\java.exe   newatlanta.sedebugger.ServletExecDebugger  -home
../ServletDebug -root .
Working Directory - D:\Work\Tutorial\
```

```
Class Path -
D:\Work\lib\Tapestry.jar;D:\Work\lib\jaxp.jar;D:\Work\lib\parser.jar;D:\Wor
k\lib\ServletExecDebugger.jar;D:\Work\lib\gnu-
regexp.jar;.;d:\tools\Kawa5.0beta1\kawaclasses.zip;C:\jdk1.2.2\lib\tools.ja
r;C:\jdk1.2.2\jre\lib\rt.jar;C:\jdk1.2.2\jre\lib\i18n.jar
New Atlanta ServletExec Debugger 2.2
    Copyright (c) 1997-1999 New Atlanta Communications, LLC.
    All rights reserved.    http://www.newatlanta.com/
ServletExec 2.2 initialized
ServletExec ServletExec listening on port 8080
```

Like any servlet engine, servlets must be configured before they can be invoked. ServletExec Debugger includes a servlet for administrating and configuring its environment. This is accessed with the following URL:

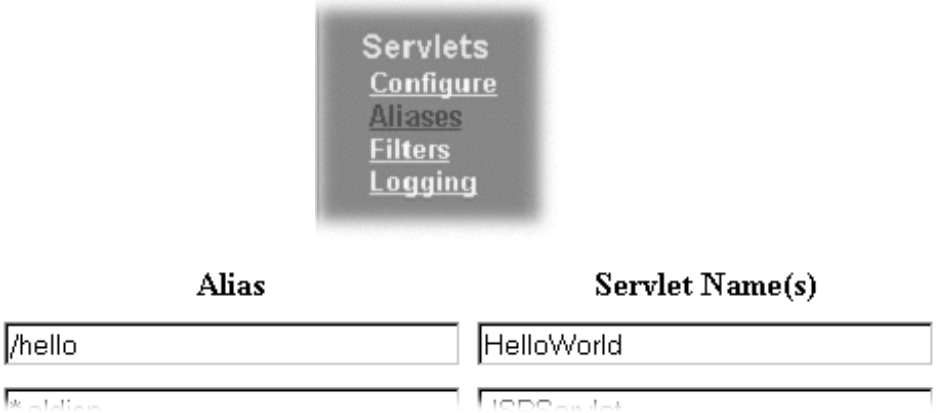
```
http://localhost:8080/servlet/admin
```

The administration interface allows several aspects of the servlet container to be managed; we're mostly intererested in mapping servlets to Java servlet classes, and to mapping URL fragments to servlets.



We need to create a new servlet named "HelloWorld" tha maps to the Java servlet class we've created:

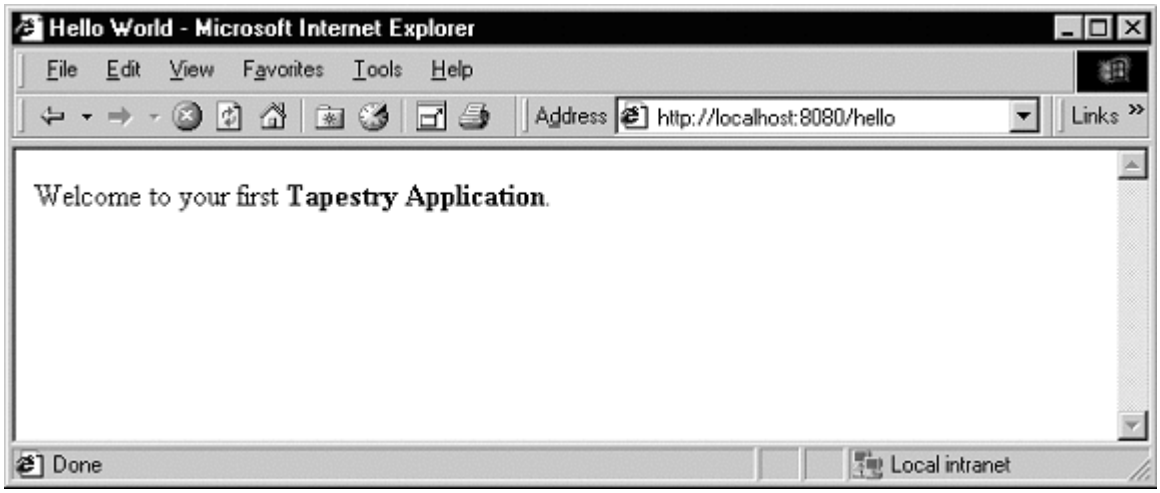
Next we need to create a URL alias for the servlet:



Finally, we can use the servlet alias to build a URL:

```
http://localhost:8080/hello
```

Which will result in the following page:



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 now 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.

Chapter

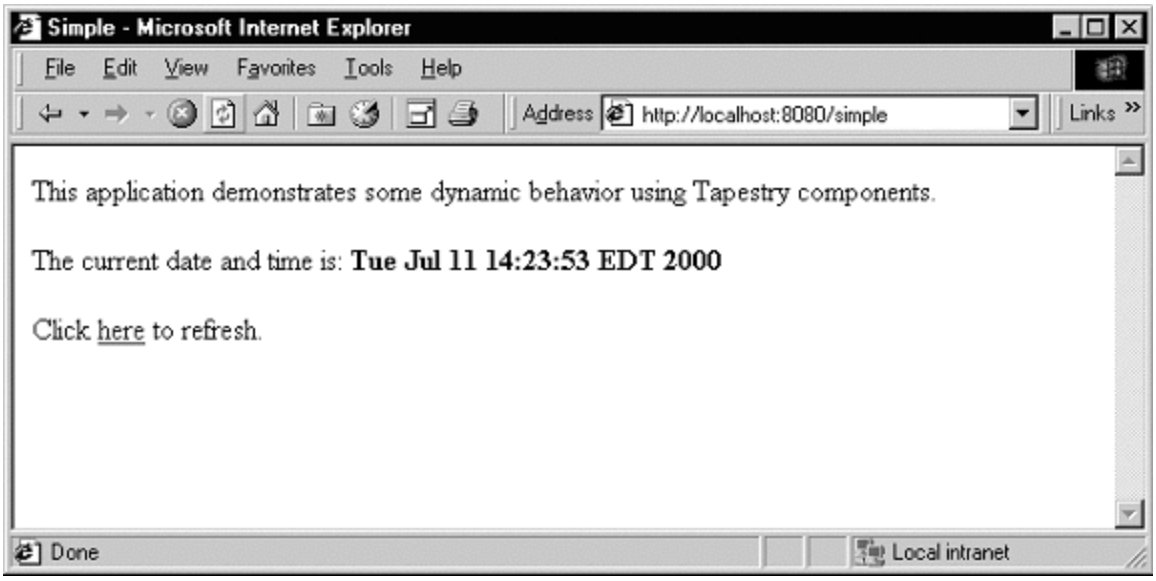
4

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 javax.servlet.*;
import javax.servlet.http.*;

public class SimpleServlet extends ApplicationServlet
{
      protected IApplication getApplication(RequestContext context)
      {
            String name = "simple.application";
            IApplication application;

            application = (IApplication)context.getSessionAttribute(name);

            if (application == null)
            {
                  application = new SimpleTutorialApplication(context,
                                                              null);
                  context.setSessionAttribute(name, application);
            }

            return application;
      }
}
```

The bold text identifies the only significant changes from the previous HelloWorldServlet class. We are storing the application specification under a different name as an attribute of the HttpSession.

SimpleTutorialApplication.java

```
package tutorial.simple;

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

public class SimpleTutorialApplication extends SimpleApplication
{
  public SimpleTutorialApplication(RequestContext context, Locale locale)
  {
    super(context, locale);
  }

  protected String getSpecificationAttributeName()
  {
    return "Simple.specification";
  }

  protected String getSpecificationResourceName()
```

```
  {
    return "/tutorial/simple/Simple.application";
  }
}
```

Again, the bold text shows the significant changes. We use a different attribute name of the ServletContext to store the parsed application specification (so it can be shared between sessions). Since this is a different application, we use a different application specification.

The application specification is also straight forward:

Simple.application

```
<?xml version="1.0"?>

<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 (highlighted in blue and underlined). "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 real 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 produce 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, it 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 "weaved" 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"?>

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

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

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

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

            <static-binding>
                  <name>page</name>
                  <value>home</value>
            </static-binding>
      </component>
```

```
</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 using 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.spec.*;
import com.primix.tapestry.*;

public class Home extends BasePage
{
  public Home(IApplication application,
              ComponentSpecification componentSpecification)
  {
    super(application, componentSpecification);
  }

  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.

To run this new Tapestry application, you'll have to map servlet 'Simple' to class 'tutorial.simple.SimpleServlet' and map alias '/simple' to servlet 'Simple'. You can then use the following URL to try out the dynamic web application:

```
http://localhost:8080/simple
```

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>Wed Jul 19 10:48:26 EDT 2000</b>

<p>Click
<a href="/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 `</a>` tags, and created an appropriate URL.

Chapter

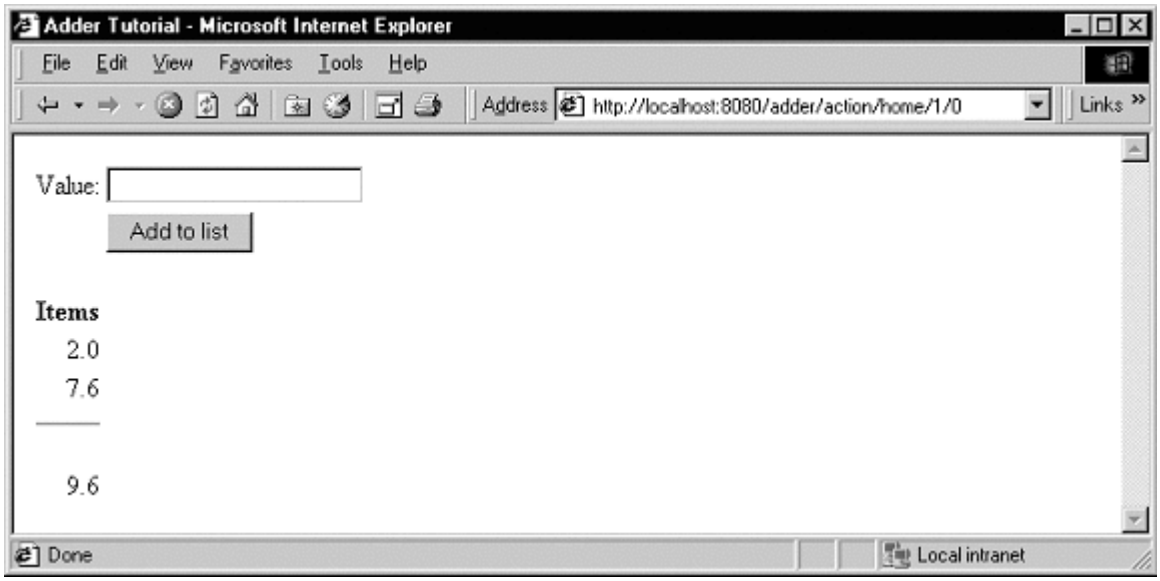
5

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 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 though the list of items, and Insert components will be used to present each item in the list, as well as the total.

As with the previous examples, the servlet and application objects are simple variations on the previous two sets (they are ommited 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="form">

<table>
  <tr>
    <td align=right>Value:</td>
    <td><jwc id="textfield"/></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 seen the Insert components.

Next we have the specification:

Home.jwc

```
<?xml version="1.0"?>

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

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

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

  <component>
    <id>textfield</id>
    <type>TextField</type>

    <binding>
      <name>text</name>
      <property-path>textFieldValue</property-path>
    </binding>
  </component>

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

    <binding>
      <name>source</name>
      <property-path>items</property-path>
    </binding>

    <binding>
      <name>value</name>
      <property-path>currentItem</property-path>
    </binding>
  </component>

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

    <binding>
      <name>value</name>
```

```
      <property-path>currentItem</property-path>
    </binding>
  </component>

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

    <binding>
      <name>value</name>
      <property-path>total</property-path>
    </binding>
  </component>

</specification>
```

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 the currentItem property of the home page (through its value parameter). That means the value will be in the home page's currentItem property when the insertCurrentValue component needs it.

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

public class Home extends BasePage
{
  private List items;
  private double currentItem;

  public Home(IApplication application, ComponentSpecification
specification)
  {
    super(application, specification);
  }

  public void setCurrentItem(double value)
  {
    currentItem = value;
  }
```

```
public double getCurrentItem()
{
  return currentItem;
}

public List getItems()
{
  return items;
}

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

  fireObservedChange("items", value);
}

public void detachFromApplication()
{
  items = 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)
      {
        if (currentItem != 0.0)
          addItem(currentItem);

        currentItem = 0.0;
      }
    };

  }

  public String getTextFieldValue()
  {
    if (currentItem == 0.0)
      return null;

    return Double.toString(currentItem);
  }

  public void setTextFieldValue(String value)
  {
    try
    {
      currentItem = Double.parseDouble(value);
    }
    catch (NumberFormatException e)
    {
    }
  }
}
```

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

In addition, 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.

Launching the Application

Run the application, then use the ServletExec Admin page to configure the servlet. Map servlet "Adder" to "tutorial.adder.AdderServlet" and map alias "/adder" to servlet "Adder".

Enter the following URL to start the application:

```
http://localhost:8080/adder
```

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**
textFieldValue	String	R / W	Converts between String and double for the currentItem property.
currentItem	double	R / W	Item being displayed or value entered in 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 enterred 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.

Persistant Page State and Page Pooling

The home page of this application uses a persistant 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 values such as the user's name and address, or (in this case) the list of numbers enterred 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 servlet, or page, or whatever was directly responsible for managing this … which leads to many half realized, ad-hoc solutions and an avalanche of bugs, not to mention, security holes.

With Tapestry, the framework takes care of these 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. Within the Tapestry framework, all of these pages, components, specifications and templates are converted into objects. Assembling a page is somewhat expensive: it involves reading all those specifications and templates, creating and initializating 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 unacceptible. 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 concievably 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 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 seperates the persistent state of a page from the actual page objects. The state is stored seperately, 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 persistant properties set by the page recorder.

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

- The property must be serializable; this includes Java scalar types (boolean, int, double, etc.), Strings, common collection classes (`ArrayList`, `HashMap`, etc.) and user-defined classes that implement `java.io.Serializable`.

- 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 persistant 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 renderred. The currentItem property takes on different values from the items List as the page is renderred. Dynamic state is easier to handle than persistant state; for completeness, it must also be reset in the `detachFromApplication()` method.

Chapter 6

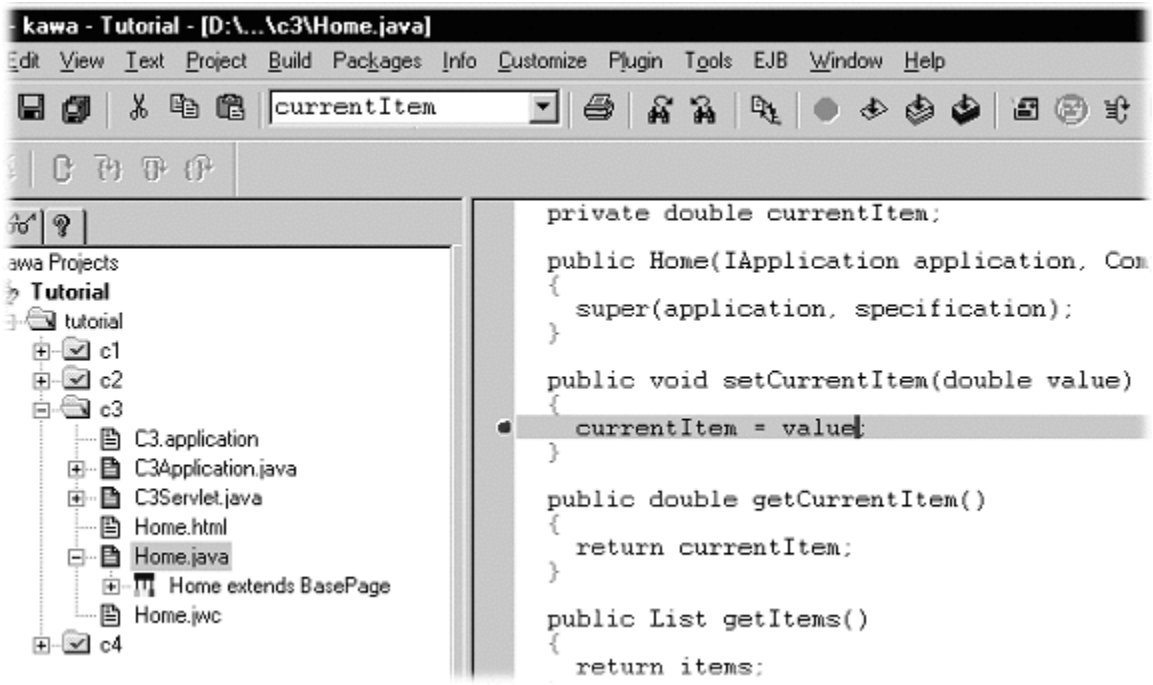
Debugging a Tapestry Application

We'll show how to debug a Tapestry application while it runs, by running the servlet container inside Kawa's debugger.

We're going to make a quick detour and discuss debugging a Tapestry application using Kawa.

We'll continue using the previous example, this time setting a breakpoint to demonstrate how the Foreach component updates the page property 'currentItem'.

First, edit the file `tutorial/adder/Home.java`. Navigate to the method `setCurrentItem()` and set a breakpoint by clicking **F9** or the menu item Build ▸ Breakpoint Set/UnSet. A red marker appears in the gutter along the left edge:



At this point, you can launch the debugger, using the Build ‣ Debug ‣ Run menu item, or by hitting **F5**, or by clicking the debug icon on the toolbar.

<<image of debug icon>>

At this point, Kawa will reconfigure itself slightly, adding a "JVMDI Watch" window (this is window that allows values to be displayed while debugging).

Launch the Adder application (from the previous chapter) with the URL:

```
http://localhost:8080/adder
```

When the form comes up, enter a value and click the submit button.

The Kawa window will raise itself, and the Project pane (along the left side of the window) will show the stack trace leading upto the break point. You can use the stack track to inspect the object, or see the parameters to the method.

Hit the continue button (or Build ‣ Debug ‣ Cont menu item, or **F5**) to allow ServletExec and Tapestry to finish the response.

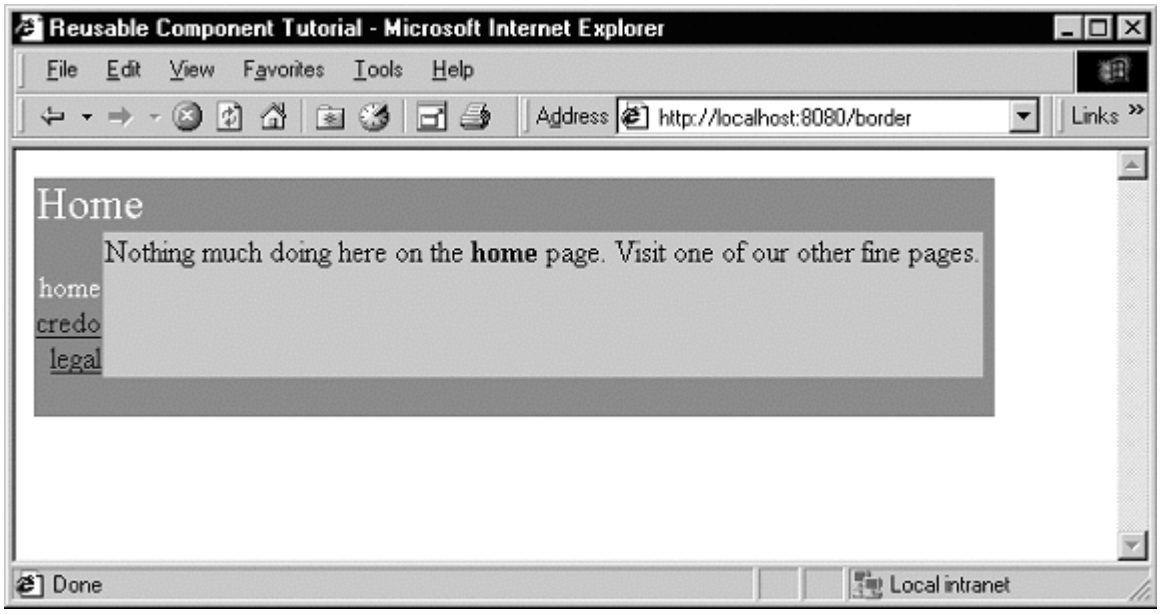
Chapter

7

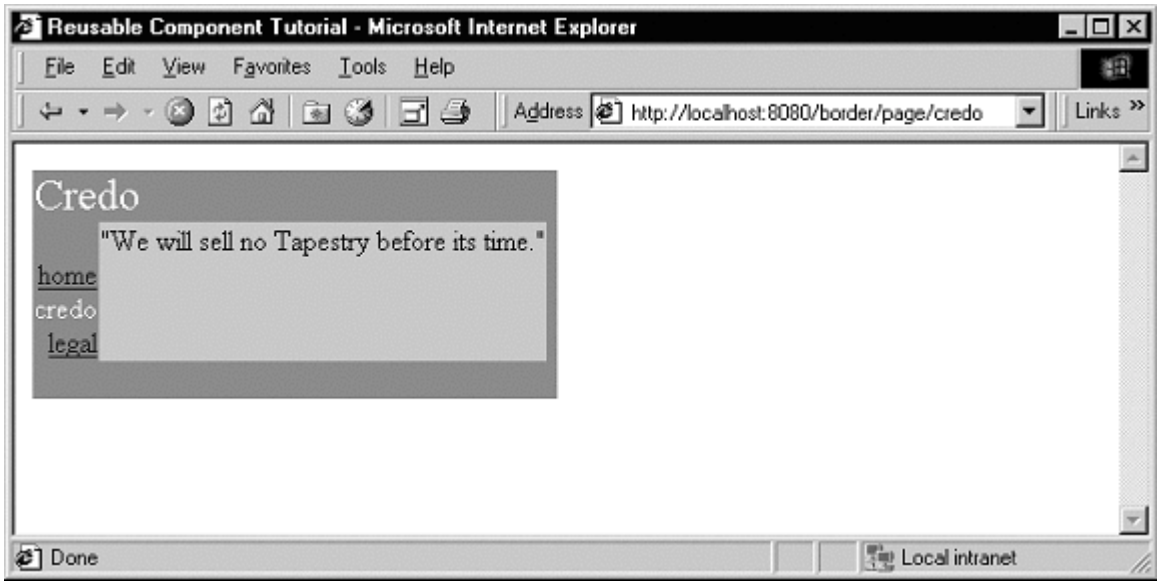
Re-usable Components

Tapestry is designed to facilitate the creation of re-usable comopnents; 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 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.



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.

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:

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 an <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). 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

```
<HTML>
<head>
<title>Reusable Component Tutorial</title>
</head>
<body>
<table border=0 bgcolor=gray cellspacing=0>
  <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 valign=top bgcolor=silver>
      <jwc id="wrapped"/>
    </td>
    <td> </td>
  </tr>
  <tr>
    <td colspan=3> </td>
  </tr>
</table>
</body>
</HTML>
```

The insertPageTitle component provides the title of the page. The e, link and insertName components provide the inter-page navigation links. 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"?>

<specification>
  <class>tutorial.border.Border</class>

  <parameter>
    <name>title</name>
    <java-type>java.lang.String</java-type>
    <required>yes</required>
  </parameter>
```

```
<parameter>
  <name>pages</name>
  <required>yes</required>
</parameter>

<component>
  <id>insertPageTitle</id>
  <type>Insert</type>

  <inherited-binding>
    <name>value</name>
    <parameter-name>title</parameter-name>
  </inherited-binding>
</component>

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

  <inherited-binding>
    <name>source</name>
    <parameter-name>pages</parameter-name>
  </inherited-binding>

  <binding>
    <name>value</name>
    <property-path>pageName</property-path>
  </binding>
</component>

<component>
  <id>link</id>
  <type>Page</type>

  <binding>
    <name>page</name>
    <property-path>pageName</property-path>
  </binding>

  <binding>
    <name>enabled</name>
    <property-path>enablePageLink</property-path>
  </binding>
</component>

<component>
  <id>insertName</id>
  <type>Insert</type>

  <binding>
    <name>value</name>
    <property-path>pageName</property-path>
  </binding>
</component>

<component>
```

```
    <id>wrapped</id>
    <type>InsertWrapped</type>
  </component>

</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 possibilites (List, Iterator, Java array) that are acceptible as the source parameter to a Foreach.

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.

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 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"?>

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

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

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

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

</specification>
```

The pages property is retrieved from the application, which implments 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 convienience.