

Tapestry Development (Pt. 1): Creating a CRUD Listing with Tapestry

Sandcast Software BrownBag Developer Series

sandcastsoftware.com/brownbag

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

1. Tutorial Introduction	2
2. Introduction	4
3. Setting up our environment	11
4. The Application	14
5. Wrap-up	37
6. Feedback and Resources	38

Section 1. Tutorial Introduction

About the BrownBag Development Series

This is a series of tutorials designed to get developers up and running with new and existing technologies. They are designed to be completed within an hour (maybe your lunch hour), but to provide enough background material for you to continue your education in this technology you are attempting to use. If you have any questions regarding any of these tutorials, have a suggestion for a new series, or are interested in writing a tutorial please contact brownbag@sandcastsoftware.com. You can always find the latest BrownBag tutorials at <http://sandcastsoftware.com/brownbag/>.

Introducing the Tapestry development series

This series of articles is designed to introduce to the Tapestry Web application framework through building a real-world application. It will also cover how to integrate other technologies with Tapestry that will aid you in your development of world-class Web applications.

Here is a brief overview of the entire series:

Creating a CRUD application (pt. 1) - In this tutorial we will walk through creating a Create, Read, Update and Delete listing using Tapestry

Enhancing our application using Hibernate (pt. 2) - In this tutorial we will walk through integrating Hibernate with our existing CRUD listing.

Enhancing our application using Spring (pt. 3) - In this tutorial we will walk through integrating the Spring framework with Hibernate and other beans that we have created with our existing application.

Enhancing our application using HiveMind (pt. 4) - In this tutorial we will walk through the same steps that we did with the Spring framework to highlight the differences between Spring and HiveMind.

Enhancing our listing with a paging component (pt. 5) - In this tutorial we again turn our eye towards the interface by implementing a paging component that allows the user to more easily navigate multiple results.

Enhancing our listing with a search component powered by Lucene (pt. 6) - In this tutorial we add a Lucene-based search component to our listing page to aid in user navigation.

Enhancing our listing with a sorting component using contrib:table (pt. 7) - In this tutorial we investigate one of the contributed components `contrib:table` and how we can use it to sort our listing on specific columns.

Enhancing our listing with hot-links (alphabetical jump-links) (pt. 8) - In this tutorial we investigate how to implement alphabetical jump-links for our listing. This would allow users to click on a letter and go to a page with only those items beginning with that letter to be shown. We will be using Lucene and a custom component to achieve this.

Creating a new listing using the contrib:tree component (pt. 9) - In this tutorial we investigate another type of listing, the tree listing and how we can achieve it using the contrib:tree component.

For this tutorial series we will be using Tapestry 3.0 for all of the examples, with the development examples using Eclipse with the Spindle plugin.

Section 2. Introduction

What is a CRUD Listing?

In this tutorial we will be creating a CRUD Listing using Tapestry 3.0. Before we start you should be familiar with Java, and creating Web applications in general. This tutorial should take between 2 and 3 hours depending on what you already have installed.

For those of you unfamiliar with a CRUD Listing let's start with the acronym first. CRUD stands for Create, Read, Update, Delete, which is a pretty standard application. Java web developers have a need for this capability in just about every application they could possibly write, so it is a widely used paradigm that every developer should know how to implement. In short here is how our application breaks down so far:

- Listing - this provides a listing of rows of items we wish to perform the actions on
- Create (or Add) - this allows the user to add a new item to the system
- Read (View or Edit) - this allows the user to view a particular item either for editing or in a read-only view
- Update - this allows the user to edit and update a particular item
- Delete - this allows the user to delete a particular item (or items)

Here is what a typical CRUD listing looks like when implemented:

	Item Title	Price	Bids	Time Left ▲
	Sun SPARCstation 20 Loaded 125MHz/256MB System *N/R*	\$9.95	1	1h 12m
	SUN SPARCSTATION 5 WORKSTATION  	\$14.99 \$19.99	- <i>Buy It Now</i>	2h 27m
	SUN SPARCSTATION 5 WORKSTATION 128MB RAM NR! 	\$26.00	2	2h 31m
	SUN SPARCSTATION 10 WORKSTATION  	\$14.99 \$19.99	- <i>Buy It Now</i>	2h 35m
	Sun Sparcstation 20 Base Unit   30 Day Warranty	\$20.00 \$25.00	- <i>Buy It Now</i>	17h 49m
	Sun SPARCstation 10/20 128MB Memory (8* 16MB)   90 Day Warranty	\$28.00 \$30.00	- <i>Buy It Now</i>	18h 04m

(Note while this is not a typical CRUD, it does reflect a typical Web listing of items).

As you can see we have a table listing with each item on its own line, users could come to this screen when managing a group of users, or when searching for an item and more than one needs to be displayed. Each row has an edit and delete link on the right (this is the typical layout, but is easily changed based on user interface testing or application needs).

Clicking on the item name typically takes a user to the view interface (as opposed to edit which brings up a form). Delete typically would take a user to a confirmation page where they can confirm their choice to delete that particular item, alternatively you could display a checkbox next to each item with an additional button that performs an action on each checked item (e.g. - Delete). Edit takes the user to a form similar (if not identical) to the Add screen but pre-filled with information on that particular item and clicking on Done or Submit updates the record in the database, or to an edit confirmation page. Finally we have the Add link which takes the user to an empty form which they can fill out and submit to save that new record to the system.

One final note about the listing, if you get too many items in the listing it will become hard for the user to navigate and find what they are looking for. Two additional items can help in this, search and paging controls. Search allows the user to narrow the criteria to determine what items are showing. Additionally, sorting and hot-links (like alphabetical jumping) can also reduce the time in finding items. Finally we have paging controls which allow the developer or user to specify how many items are displayed on a page at a time and let them navigate through the different pages using the navigational page controls. All of these topics will be addressed in the series.

In the interest of keeping this tutorial focused on using and teaching Tapestry we will not be employing a database or other technologies to assist us. Later tutorials in the series will cover Hibernate, Spring, HiveMind, Lucene, and specific Tapestry components. But we will be using industry best-practices so that when we add in other technologies it will be as fluid as possible.

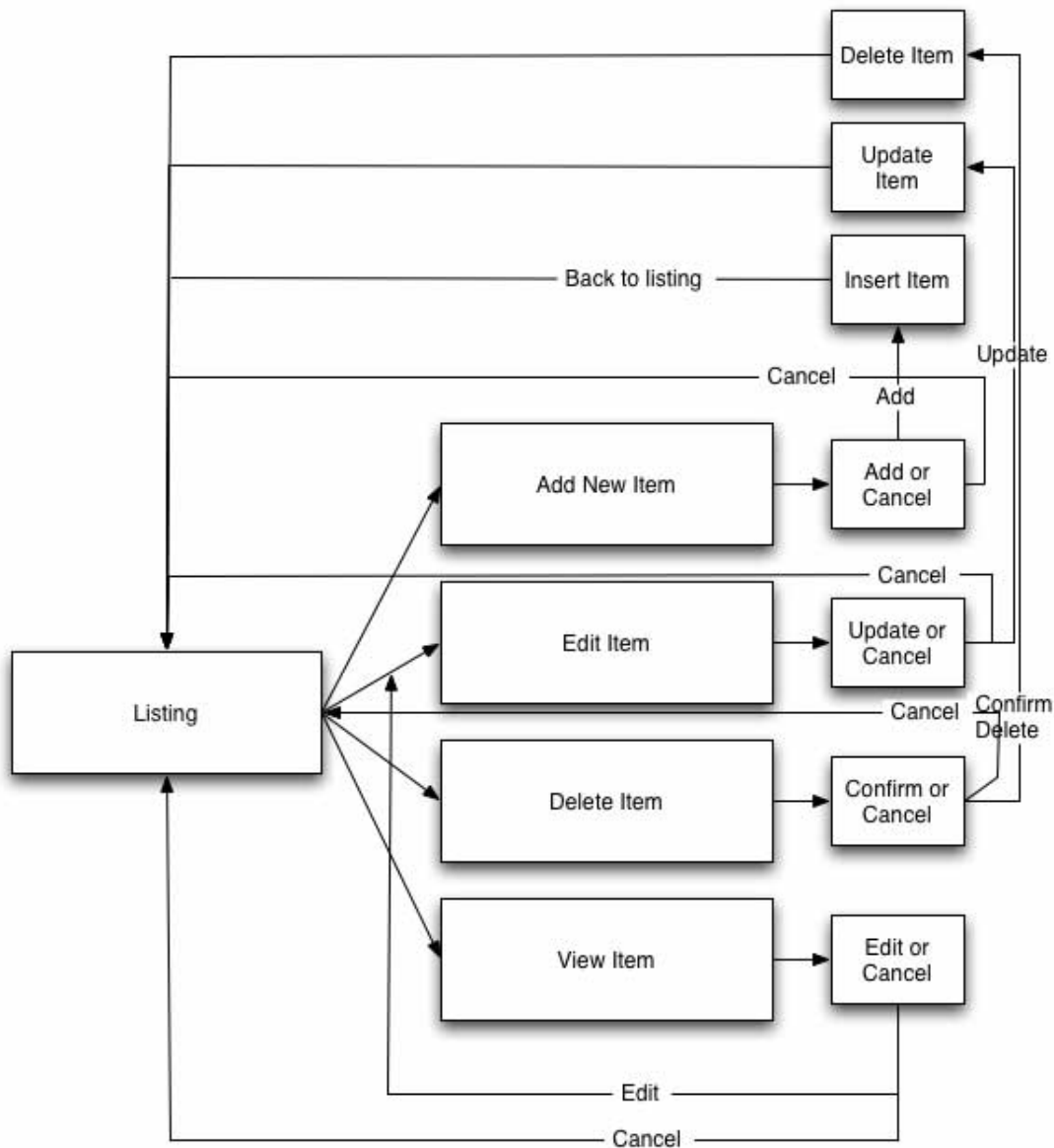
The application flow (the concept)

Ok, let's get started. The first thing we want to do is to figure out what our application flow will be for this mini-application.

We've discussed this a little bit in the previous section, but we can take it action-by-action to make sure that everything is clear.

- **Add** - Once the user clicks on the Add button/link they are presented with a form, clicking on Cancel takes them back to the listing with no action being taken, clicking on Add (or Submit, Insert, etc.) inserts the item into the system and returns the user to the listing.
- **Edit** - Once the user clicks on the Edit button/link they are presented with a form representing the particular item, again clicking on Cancel takes them back to the listing with no action, clicking on Update updates the system with the user's changes and returns them to the listing.

- **Delete** - Once the user clicks on the Delete button/link they are presented with a confirmation page identifying the item(s) they wish to delete, cancel takes them back to the listing, confirming deletes the item(s) and returns the user to the listing.
- **View** - Once the user clicks on the item link (or you can have a separate View link) they are presented with a read-only version of the selected item and can either Cancel or Edit that item. Cancelling returns them to the listing, Edit takes them to an Edit form and follows the normal flow from there.



As you can see it can get rather complex quickly, one piece that we didn't add in here is form validation, we will be adding in validation later in this tutorial but we won't add it into the flow just yet. Additionally, we are only going to concern ourselves with the Add, Edit, and Delete portions, once you finish you should see that it wouldn't be difficult to add in a View piece to your application should you need it.

Introducing Tapestry (the technology)

Why have we chosen Tapestry to implement our CRUD Listing? There are several reasons why one might choose Tapestry over some of the other Web application frameworks currently available and a few are listed here, but first who's the competition?

- **Struts** - this is probably the biggest competition any Web app framework can face. First, it is kind of "blessed by Sun" as the original author and primary committer, Craig McClanahan is a Sun employee and on a few JCP committees regarding Sun j2ee technologies. It is a Model-View-Controller style app framework that utilizes JavaBeans as the Model, JSP as the primary View (Velocity templates are also supported externally), and a custom Servlet as the controller.
- **WebWork2** - this is another MVC style app framework that utilizes JavaBeans as the Model, several different View technologies (JSP and Velocity are primary), and a custom Servlet as the controller.
- **JavaServer Faces (JSF)** - the new kid on the block from Sun takes a different approach (very similar to Tapestry) in that it gets away from the MVC paradigm and shifts more to a componentized view of things. This is a very flexible way of creating an application and more object-oriented in nature.
- For more frameworks you can look at the Wafer Project (<http://waferproject.org>) to see their capabilities

Why would we choose Tapestry over these other technologies (especially JSF)?

- **Maturity** - Tapestry has gone through many revolutions already and the solutions are tested in real environments. There are more changes coming, but it is based on real-world situations not design by committee.
- **Object-Oriented** - the solution that Tapestry offers is much more flexible than a traditional MVC Web app and allows for component re-use in multiple projects.
- **Community** - Tapestry has an active and helpful community in which to ask questions.
- **Growth** - Interest in Tapestry is growing which means more users and developers will become involved with it to help improve it.

Why might we choose something else?

- **Management dictates** - Management wants to use a "popular" or commercially supported framework, such as Struts or JSF.
- **Commercial support** - You and your team are looking for commercial support in tools such as WSAD (WebSphere Application Developer) or other such Web application servers.
- **MVC** - You or your team are specifically looking for a MVC based Web app framework.

Brief Overview of Tapestry

Tapestry is about different objects working together; Tapestry pages contain properties that Tapestry components read and update. OGNL (Object Graph Navigation Library) is an expression language that tie pages and components together.

In Tapestry the biggest concept to learn is that just about everything is a component.

What is a component?

A component is a wrapper around a specific piece of functionality, be that a button, a form element, a form, or a page. Each component can accept passed in parameters, which may change it's behavior, the model that it binds to, or what it displays. In Tapestry these parameters are passed in by reference rather than value so it is possible to modify these values in the component that accepts them.

Here's a simple example of a HTML text field component used in forms:

```
<span jwcid="inputId@TextField" value="ognl:visit.id"
size="8"/>
```

This tells Tapestry that you want to create a new instance of a TextField component and bind it to the value of `getVisit().getId()` with the size of "8". Alternatively you could code the same example using form elements - `<input type="text" jwcid="inputId@TextField" value="ognl:visit.id" size="8"/>` , the only piece that Tapestry cares about is the `jwcid`, which defines what component you are using. Each individual component type defines specific parameters that you can pass into the component that are not required, such as `hidden` (which turns the TextField into a password field). Other components have much more complex parameters that take objects like a `java.util.Date` object, that the component will operate on and bind to. While we will not be covering OGNL expressions in depth in this tutorial we will cover some of the basics of an OGNL expression as they are used throughout our application.

What is binding?

Binding is the process of tying a form change with an actual data object that you are presented with when processing the form. This differs from frameworks like Struts where you have to bind forms with FormBeans, in Tapestry you can bind an object to any element in the form as long as you can get access to it through the page. We will cover this in greater detail later.

Other properties of components

Components can also contain other components as well, which is a good thing otherwise we wouldn't be able to place a form in a page, or be able to create site templates using the Border component.

Components can also disregard items in their body. What does this mean? Anything in-between the beginning of a component tag and the end will be discarded. Why would we do this? This makes it easier to hand off a modified HTML template to a designer for them to modify and place inside of the component tag test data enabling to see what it will look like at design time. Rather than trying to figure out JSP and tag libraries, they are dealing with HTML, which they can modify as needed.

Getting Started

To get started with this tutorial the easiest thing to do is to download the source code from the site at <http://sandcastsoftware.com/downloads/brownbag/tapestry/tutorial1.zip> (or .tar.gz, see the listing for the appropriate download). You can use either Maven or Ant to build the project, Maven is highly recommended as it provides more functionality (the Ant build was created using Maven ;-). Maven can be downloaded from <http://maven.apache.org>, Ant can be downloaded from <http://ant.apache.org>, a familiarity with either of these will be required for this tutorial, a Maven tutorial will be provided later for those interested in learning about it.

Once you have downloaded and unpacked the tutorial source and setup either Maven or Ant you can get things started. Eclipse is the assumed platform of development (3.0 is recommended). Go ahead and fire up Eclipse and setup a new project for this tutorial pointing at the unpacked directory. If you want Maven to create your Eclipse .classpath and .project files type `maven eclipse` in the root directory of the unpacked source, you will see Maven download all the necessary jar files and then create your .classpath and .project files for you. If you have done this you will need to do one final step to tell Eclipse where all of your new jar files are at. In Eclipse go to Window > Preferences, then choose Java > Build Path > Classpath variables. Click New and type in `MAVEN_REPO` and point it at your local Maven repository (typically in your User Home directory under `.maven/repository`, unless you have changed it). Refresh your project and voila! For a more detailed exploration into Maven you can check out my chapter on Maven in the [Professional Java Tools for Extreme Programming](#) by Wrox Press.

Directory Structure

Let's familiarize ourselves with the project's directory structure a bit so we know where all the files are going to be.

```
sc-tapestry-crud1
  src
    java
    test
    webapp

  target
    classes
```

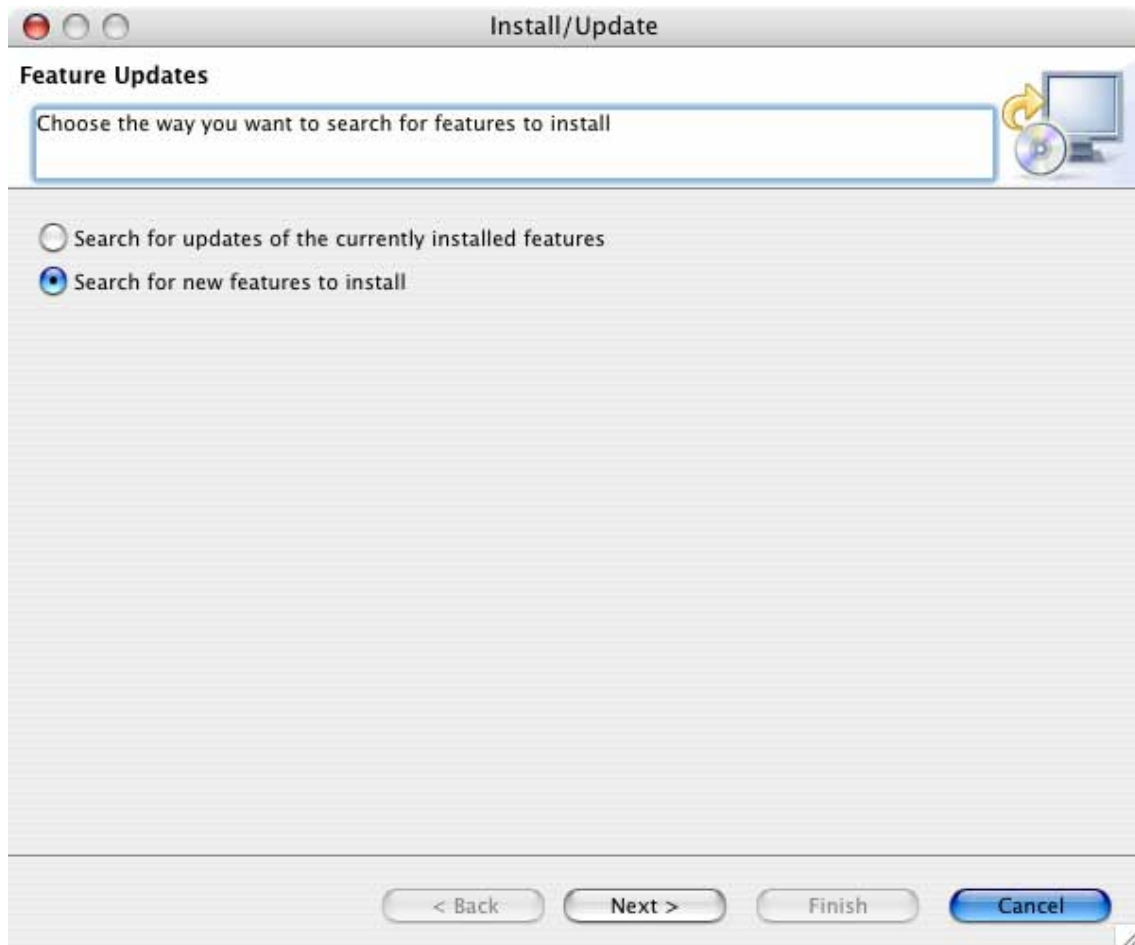
```
test-classes  
sc-tapestry-crud1.war  
sc-tapestry-crud1
```

This is a pretty simple setup, `src/java` contains all of our java files for the application, `src/test` contains our unit tests and `src/webapp` contains all the extraneous files for our final web application. When we are ready to build Maven will compile the java files to `target/classes`, compile and run the unit tests to `target/test-classes` and then when we create our web app Maven will create the `sc-tapestry-crud1` folder place all the classes and other files from webapp into there and then package it as a war file.

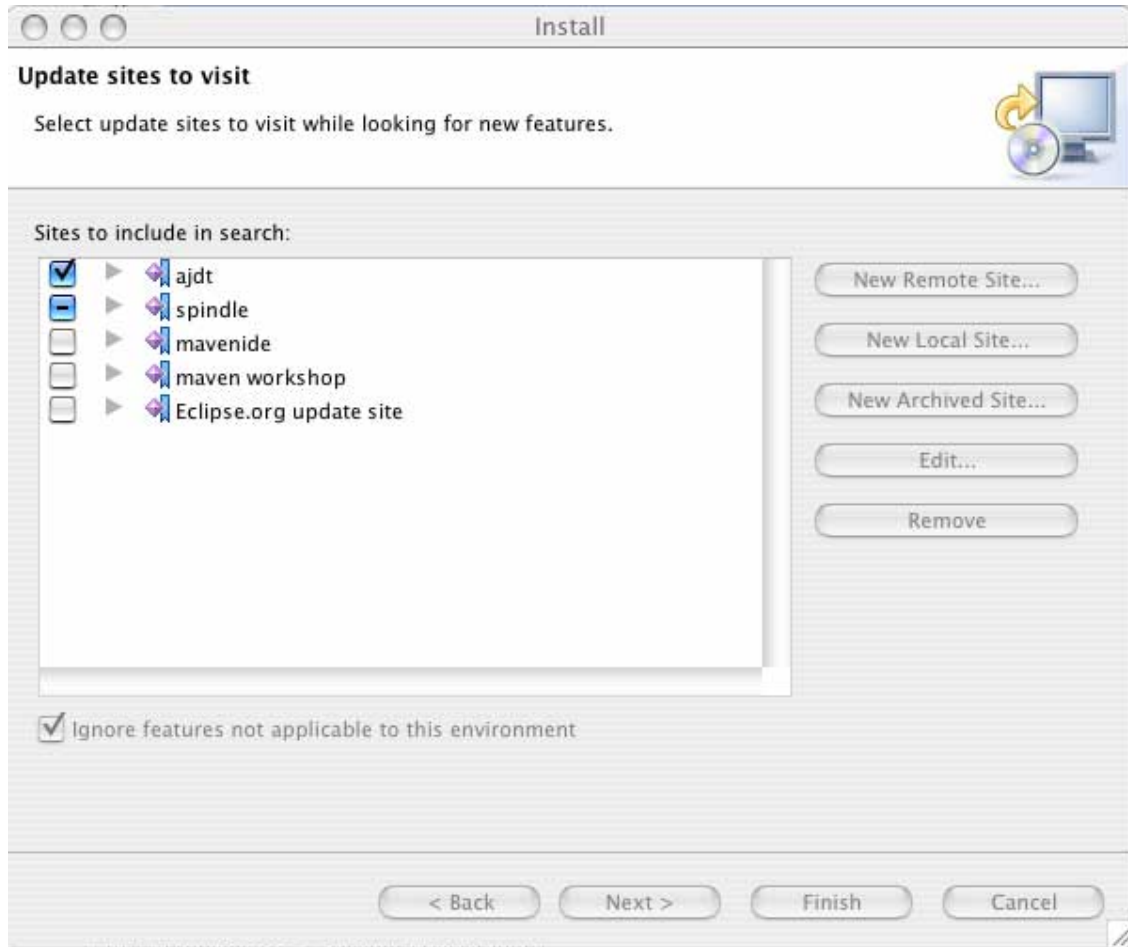
Section 3. Setting up our environment

Spindle

Now that we have the project started we need to go ahead and setup our Eclipse environment. The first thing we will need to do is to install the Spindle plugin if you haven't already done so. To download the latest and greatest click on the Help menu and then Software Updates > Find and Install. You will be presented with a screen similar to the one below:



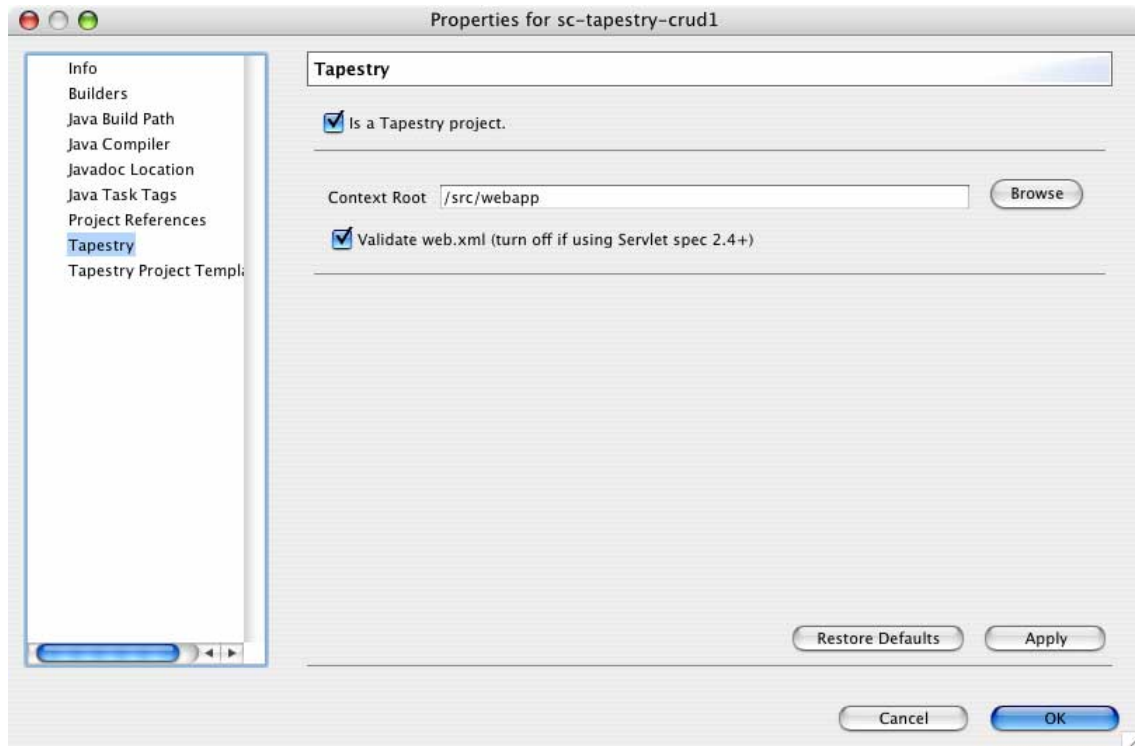
Choose "Search for new features to install" and click Next. You should then see a screen similar to this:



On this screen we want to create a new archived site, go ahead and call it Spindle and then paste in this url - <http://spindle.sourceforge.net/unstable/>, You will then be presented with a list of plugins available, go ahead and select the latest Spindle (as of this writing 3.1.16) and if you want the Jetty plugin (an embeddable Web application server) you can do that as well.

Once you've finished installing Spindle go ahead and let Eclipse restart.

Now that we've installed Spindle we need to tell Eclipse that what we are building is a Tapestry project. Right-click on the main project name and choose properties. You should see something like this:



Choose Tapestry on the left and then you can fill in the values as you see here, check off the fact that it is a Tapestry project and the web application context is src/webapp and if you want you can have Spindle validate your web.xml.

Section 4. The Application

The DVD Object

Let's start designing the application to understand what we will need to create in the way of classes. What do we want to create a listing of? Let's start with a listing of DVDs. What we want to do then is define what a DVD object looks like.

Since this will be a simple example (at least at first) we want to keep our object model relatively simple as well. At it's root, what does a DVD have? A DVD has a Title, a Cost, and a Cast (people associated with the film). Here's what it would look like in Java:

```
public class DVD {

    private int id;
    private String title;
    private float cost;
    private ArrayList cast;

    /**
     * @return Returns the id.
     */
    public int getId() {
        return id;
    }

    /**
     * @param id The id to set.
     */
    public void setId(int id) {
        this.id = id;
    }

    /**
     * @return Returns the cast.
     */
    public List getCast() {
        return cast;
    }

    /**
     * @param cast The cast to set.
     */
    public void setCast(List cast) {
        if(null != cast) {
            this.cast = new ArrayList(cast);
        }
    }

    /**
     * @return Returns the cost.
     */
    public float getCost() {
        return cost;
    }
}
```

```
/**
 * @param cost The cost to set.
 */
public void setCost(float cost) {
    this.cost = cost;
}

/**
 * @return Returns the title.
 */
public String getTitle() {
    return title;
}

/**
 * @param title The title to set.
 */
public void setTitle(String title) {
    this.title = title;
}
}
```

That's pretty much it, the DVD object is nothing more than a POJO (Plain Old Java Object) that stores the information we need and provides access to those properties through getters and setters. Now that we have our base object defined we need to add a way to create, edit, and save that object in a persistent way.

The DAO Pattern

Although we will not be using a real persistence layer in this first tutorial we will still be acting as if we were, and the best way to do that is through the Data Access Object Pattern. This pattern is a way to provide access to POJOs that will be persisted, either in a database, an XML file, whatever. This suits our purposes exactly, and we will do it in such a way that we can easily swap it out, by using interfaces.

The first interface we will need to define then will be the DvdDAO interface:

```
public interface DvdDAO {

    public DVD getDVD(int id);
    public void saveDVD(DVD dvd);
    public void removeDVD(DVD dvd);
    public List getAllDVDs();

}
```

This interface (once implemented) will allow us to find a given DVD object through its `id` value, we can save a DVD by passing it into the `saveDVD` method, remove a DVD by passing it into the `removeDVD` method or retrieve all of the stored DVDs by calling `getAllDVDs()`. This interface will not change

much between implementations, now we need to do a custom implementation of it for our first example.

The best way to store our information is just as a `List` internal to the `DvdDAO` implementation, which we will call `DvdDAOSimple`. The code can be seen below for this implementation, nothing too exciting going on, but two things should be noted. First, we need an easy way to store the current id of the last saved element (to emulate an auto-incrementing save) so we use a `currId` value. Second, we are using a synchronized list, if this application is accessed by more than one person we don't run into the problem of mismatched lists.

```
public class DvdDAOSimple implements DvdDAO {

    private List dvdList;
    private int currId = 0;

    /* (non-Javadoc)
     * @see com.sandcast.examples.tapestry.crud.dao.DvdDAO#getDVD(java.lang.Integer)
     */
    public DVD getDVD(int id) {
        if (dvdList == null) {
            return null;
        } else {
            Iterator iter = dvdList.iterator();
            while(iter.hasNext()) {
                DVD dvd = (DVD)iter.next();
                if(dvd.getId() == id) {
                    return dvd;
                }
            }
            return null;
        }
    }

    /* (non-Javadoc)
     * @see com.sandcast.examples.tapestry.crud.dao.DvdDAO#saveDVD(com.sandcast.examples.tapestry.crud.dao.DVD)
     */
    public void saveDVD(DVD dvd) {
        if(null == dvdList) {
            dvdList = Collections.synchronizedList(new ArrayList());
        }

        Iterator iter = dvdList.iterator();
        boolean found = false;
        while(iter.hasNext()) {
            DVD temp = (DVD)iter.next();
            if(temp.getId() == dvd.getId()) {
                temp.setCost(dvd.getCost());
                temp.setTitle(dvd.getTitle());
                temp.setCast(dvd.getCast());
                found = true;
            }
        }
        if(!found) {
            int newId = ++currId;
            dvd.setId(newId);
            dvdList.add(dvd);
        }
        return;
    }
}
```

```
    }

    /* (non-Javadoc)
     * @see com.sandcast.examples.tapestry.crud.dao.DvdDAO#removeDVD(com.sandcast.examples.tapestry.crud.dao.Dvd)
     */
    public void removeDVD(DVD dvd) {
        if(null != dvdList) {
            dvdList.remove(dvd);
        }
    }

    /* (non-Javadoc)
     * @see com.sandcast.examples.tapestry.crud.dao.DvdDAO#getAllDVDs()
     */
    public List getAllDVDs() {
        if(null == dvdList) {
            dvdList = Collections.synchronizedList(new ArrayList());
        }
        return dvdList;
    }
}
```

The user interface

Whew, the hard part is over, well only partly true. As you'll see in the source tree there are quite a few Unit tests for the DAO implementation, these were written as the functionality of the DAOSimple was being written so that a change didn't break the existing pieces. This is something that you should do for all of your projects, additionally tests should be written for any object that does anything more than act as a POJO. As soon as you start to do any modifications of the properties based on what gets passed in then it's time for a Unit test, additionally if you add in methods to the POJO that perform other functions, it's time for a Unit test.

What do we want to show on our listing? This will most likely be the first or the second thing that the user sees so we should decide what they can see in order to determine if they are looking at the right item. We don't have that many items to show, so it's simply a matter of determining what should be there for right now. How about Title and Cost? It would be difficult to show the cast on a listing page so that's out, if we had a year field then that would be a candidate. Here is what our interface will look like (It isn't pretty, but we will improve it as we move through the tutorials).

[Show DVDs](#)

Title	Cost	Edit	Delete
The Last Samurai	19.99	Edit DVD	Delete DVD

[New DVD](#)

Next comes the Add and Edit screens, for right now we are just going to allow someone to input the Title and Cost of a DVD as seen here:

[Show DVDs](#)

Entry Title

DVD Price

Now we're going to go ahead and create our user interface to interact with the backend components we have created so far.

Setting up the Web descriptor for Tapestry

The first thing we need to do is to setup our Web application. In the src/webapp directory make sure that you have created a WEB-INF directory that we can use. Whichever way is most comfortable to you create a new file called web.xml inside that WEB-INF directory to control the Tapestry servlet. Here is what it should look like:

```
<?xml version="1.0"?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_3.dtd">
<web-app>

  <display-name>Tapestry CRUD Example Application</display-name>
  <filter>
    <filter-name>redirect</filter-name>
    <filter-class>org.apache.tapestry.RedirectFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>redirect</filter-name>
    <url-pattern>/</url-pattern>
  </filter-mapping>
  <servlet>
```

```
<servlet-name>crud</servlet-name>
<servlet-class>org.apache.tapestry.ApplicationServlet</servlet-class>
<load-on-startup>0</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>crud</servlet-name>
  <url-pattern>/app</url-pattern>
</servlet-mapping>

<session-config>
  <session-timeout>5</session-timeout>
</session-config>

</web-app>
```

This tells the application server that we want all requests that go to app to go to the Tapestry ApplicationServlet instance. We have also defined a redirect filter so that we don't need to create a redirect page in the root of our web application to send it to the right place.

What happens when a request is made to
`http://localhost:8080/sc-tapestry-crud1/?`

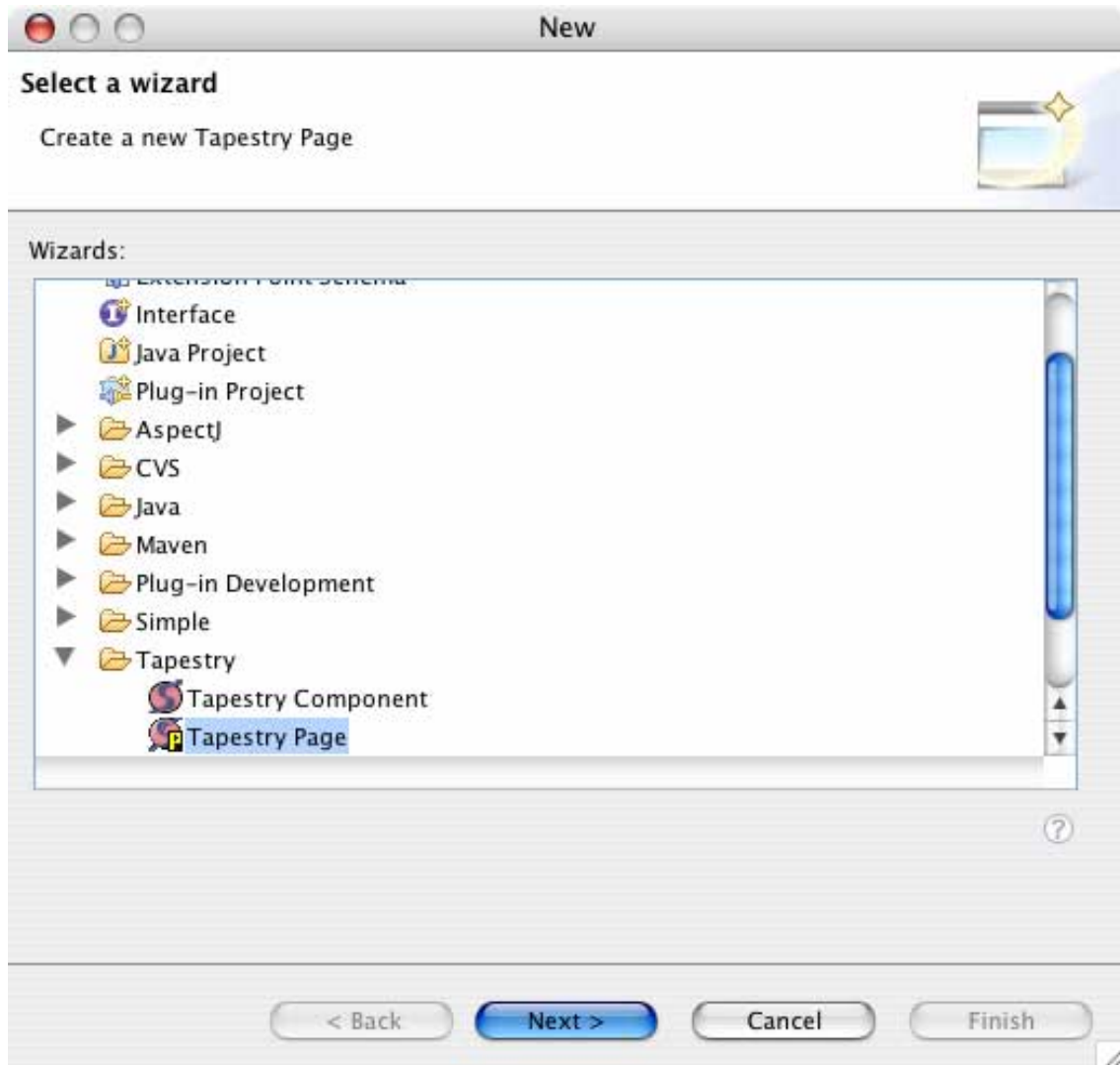
First the redirect filter intercepts the request and redirects it to the Tapestry servlet that resides at
`http://localhost:8080/sc-tapestry-crud1/app`, Tapestry then picks up that request and looks for the Home.page implementation (which we will investigate in the next section). After this initial request all application-specific requests will be under the /app context.

Next we need to start creating our actual Tapestry pieces.

Creating the Home page

Every Tapestry application relies on the Home page, you can think of this like an index page for your site, without it users wouldn't know where to start. One thing to note here is that the Home page actually needs to be called "Home", Tapestry expects that there be a start page with that name available to it on startup.

To create the Home page right-click on the src/webapp directory and choose New > Other. When the Other screen comes up choose Tapestry > Tapestry Page.



Choose Next after you have picked Tapestry Page to be presented with a series of options.

New Tapestry Page Component

Create a new Tapestry Page file resource.

Page Name:

Project:

Namespace:

File Generation:

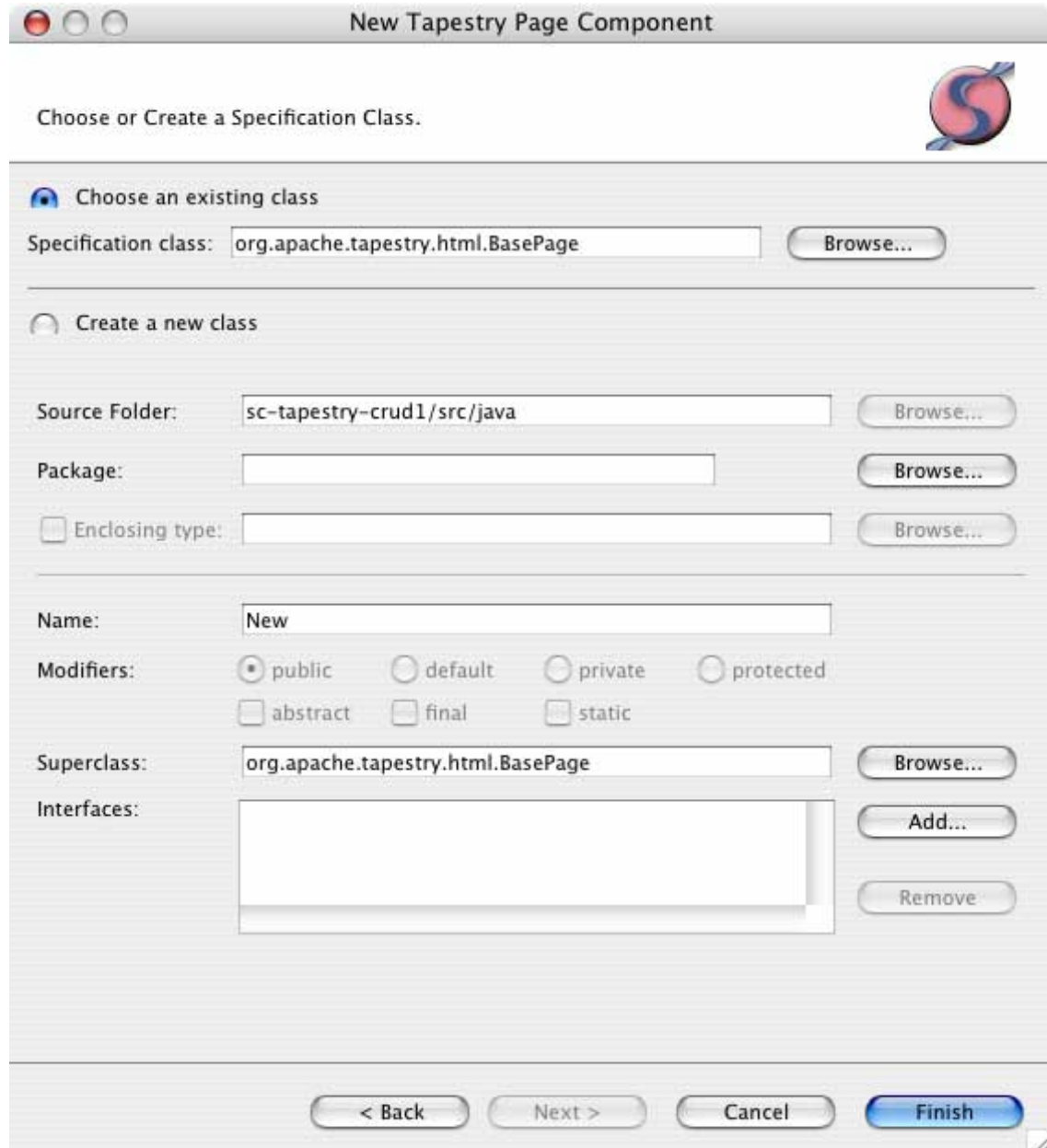
.page template:

Generate an associated HTML File?

.html template:

Choose a class for the specification on the next page...

On this screen we want to put in the page name of Home, ensure that the project is the right project. Additionally we want to leave the defaults for the .page template and also check off "Generate an associated HTML File" and leave that option as default. In the next screen we have the option of choosing what class our Home page will extend (or create our own).



Now we have some custom stuff we will need to do so we want to create a new class, go ahead and place that class in `com.sandcast.examples.tapestry.crud.pages`. Also go ahead and call the Java class name the same as the page (Home) to make things easier. Although this last step isn't absolutely necessary as we can associate a `.page` binding with any Java class we want to or none at all, in this case it just makes things clearer.

Once we've done this those files should have been opened up in the Eclipse editor. Most of these files are just skeleton files we will need to add in the bits to make them work properly.

Since Web applications are all about the interface let's start with the `Home.html` file first.

```

<span jwcid="@Border">
<table cellpadding="6">
  <tr>
    <td>Title</td>
    <td>Cost</td>
    <td>Edit</td>
    <td>Delete</td>
  </tr>
  <tr>
    <td colspan="4"><hr></td>
  </tr>
  <tr jwcid="@Foreach" source="ognl:dvds" value="ognl:dvd" element="tr">
    <td><span jwcid="@Insert" value="ognl:dvd.title"/></td>
    <td><span jwcid="@Insert" value="ognl:dvd.cost"/></td>
    <td>
      <a jwcid="@DirectLink" listener="ognl:listeners.dvdEditAction"
        parameters="ognl:dvd.id">Edit DVD</a>
    </td>
    <td>
      <a jwcid="@DirectLink" listener="ognl:listeners.dvdDeleteAction"
        parameters="ognl:dvd.id">Delete DVD</a>
    </td>
  </tr>
  <tr>
    <td colspan="4"><hr /></td>
  </tr>
  <tr>
    <td colspan="4"><span jwcid="@DirectLink" listener="ognl:listeners.dvdNewAction">New DVD</span>
  </tr>
</table>
</span>

```

It looks like HTML, for the most part. That is one of the strengths of Tapestry is that it is relatively easy for designers to get up to speed doing what they do best, design and layout. What are some of these pieces that aren't HTML though?

Attributes

- **jwcid** - This stands for java web component id (jwcid) and tells Tapestry that this is a component id that we need to look for and resolve
- **source** - This tells the Foreach component what to loop over
- **value** - This tells the component what property you want to store the current element in
- **listener** - This identifies a listener method, implemented in your page class, to be invoked when the HTML link is clicked
- **parameters** - This tells Tapestry what values you want to pass into the component when generating this link

Components

- **Border** - This is a custom component that we will need to create, it is a wrapper around the page content, providing a basic look and feel or application navigation. While not a required component, Border is a common best-practice to assist in component reuse.
- **Foreach** - As you might guess this is a component that loops over a Set, List, any type of Collection, or Array and stores the value of the current item in the loop into another page property, for access by other components.

- **Insert** - This allows us to insert dynamic content into an HTML page.
- **DirectLink** - This is one of Tapestry's link components that allows us to invoke a listener method in our page class identified by the "listener" attribute.

What's missing here?

Well, we don't have the typical HTML headers or footers, that's because it's all stored in the Border component, which we will get to a little later. Also, there's all these ognl: things floating about, what do these do?

As we stated earlier, this tells Tapestry to bind a page property with a Java property. For example `source="ognl:dvds"` would bind to a method `getDvds` that returns some kind of list, which we will see in just a minute.

The Home.page file

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification
  PUBLIC "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
  "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification class="com.sandcast.examples.tapestry.pages.Home">
  <property-specification name="dvd"
    type="com.sandcast.examples.tapestry.crud.bo.DVD"/>
</page-specification>
```

This tells Tapestry that this "page" is bound to the Java class `com.sandcast.examples.tapestry.pages.Home`, as stated earlier you can change this to any Java class as long as it extends the `BasePage` class. Additionally we have a `property-specification` here. This ties the current `dvd` in the `Foreach` to an abstract method called `getDvd` and `setDvd`. We'll get into abstract properties in the next section, but by placing the property here we are telling Tapestry that we want the framework to manage it for us.

Home.java

This is the final piece for the Home page, let's discuss this a bit before we go into the code. What do we want to do on the Home page?

- List all DVDs
- Allow the user to Edit a DVD
- Allow the user to Delete a DVD
- Allow the user to Create a new DVD

Here is how we accomplish this:

```
public abstract class Home extends BasePage {

    public abstract void setDvd(DVD dvd);
    public abstract DVD getDvd();
}
```

```

public List getDvds() {
    Global global = (Global) getEngine().getGlobal();
    DvdDAO dvdDAO = global.getDvdDAO();
    List dvds = dvdDAO.getAllDVDs();
    return dvds;
}

public void dvdEditAction(IRequestCycle cycle) {
    EditDVD nextPage = (EditDVD) cycle.getPage("EditDVD");
    Object[] parameters = cycle.getServiceParameters();
    int id = ((Integer) parameters[0]).intValue();
    Global global = (Global) getEngine().getGlobal();
    DvdDAO dvdDAO = global.getDvdDAO();

    DVD dvd = dvdDAO.getDvd(id);
    nextPage.setDvd(dvd);
    nextPage.setNewDvd(false);
    cycle.activate(nextPage);
}

public void dvdNewAction(IRequestCycle cycle) {
    EditDVD nextPage = (EditDVD) cycle.getPage("EditDVD");
    nextPage.setDvdId(0);
    nextPage.setNewDvd(true);
    cycle.activate(nextPage);
}

public void dvdDeleteAction(IRequestCycle cycle) {
    Object[] parameters = cycle.getServiceParameters();
    int id = ((Integer) parameters[0]).intValue();
    Global global = (Global) getEngine().getGlobal();
    DvdDAO dvdDAO = global.getDvdDAO();

    DVD dvd = dvdDAO.getDvd(id);
    dvdDAO.removeDVD(dvd);
    cycle.activate("Home");
}
}
}

```

The first question might be, why is the class `Abstract`? The Tapestry framework provides the ability at run-time to extend your class and provide or override accessor methods, hence the `public abstract void setDvd(DVD dvd)`; which we tie into through the page specification as we saw above. Alternatively you don't even need this method here (if you don't need to access it through code) and Tapestry will add it at run-time using the `Javassist` bytecode library (see Resources). Any property that you have named in the `.page` file can be an `Abstract` property in the Java file as long as you have made the class itself `Abstract`.

Why would we want to make these properties abstract? By specifying abstract accessor method you are allowing Tapestry to manage the transient and persistent state of your application, and is vastly preferable to JSPs which require direct access to the `HttpServletRequest` and `HttpSession` objects to accomplish the same thing.

Then we have the `getDvds()` method that binds to the OGNL expression we saw in the HTML page as `ognl:dvds`, OGNL does our handy name conversion for us and finds the correct binding method. In this method we are

then gaining access to the Global object which we will define shortly. This object allows us access to our DAO to perform all the functions that we need to on the DVD object.

Next we see our three actions that we defined as listeners on the HTML page, `dvdEditAction`, `dvdNewAction`, and `dvdDeleteAction`. To be a listener they must match the signature of `public void methodName(IRequestCycle cycle)` for Tapestry to recognize it.

Inside the edit and delete listeners we gain access to the DAO implementation through the Global object mentioned above.

The Global object

Let's dive into the Global object a bit since we will be using it so much and have only briefly mentioned it.

What is the Global object?

In short the Global object is anything you want it to be. Tapestry doesn't force you to implement a specific interface in order for a global object to be created. In fact you don't even need to call it Global, you can call it whatever you want (but Global makes it easier). One of the reasons why it is called Global though is that you can gain access to it throughout the framework, which makes it ideal for things like DAOs and other items (which we will see later). Also, it is not tied to a particular user's session (so beware if you think it does) that type of access is handled through the Visit object (again, quite similar to Global as it can be any object you want), we will cover the Visit concept later in the series.

Now, here's our code for the Global object:

```
public class Global implements Serializable {
    DvdDAOSimple dvdDAO = null;

    public DvdDAO getDvdDAO() {
        if(dvdDAO == null) {
            dvdDAO = new DvdDAOSimple();
        }
        return dvdDAO;
    }
}
```

First, why did we make this class Serializable? This ensures that should your application be run across multiple containers Tapestry will pick up the same Global object and not create a new instance of it (and throw off our list as it would be different in each container). Also notice that we don't return a `DvdDAOSimple` we are actually returning the interface, this is because we don't want to be tied to a particular implementation we want to be tied to the interface. This class will grow in later tutorials, but for right now this is all we need it to do.

Now, how do we tell Tapestry about it?

Tapestry app configuration

Every Tapestry application is configured through a `.application` file, this file is named after the servlet name we used in the `web.xml` file, in our case it will be `crud.application` and it will reside in the `WEB-INF` directory.

```
<?xml version="1.0"?>
<!DOCTYPE application PUBLIC
  "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
  "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<application name="Tapestry CRUD Example"
  engine-class="org.apache.tapestry.engine.BaseEngine" >
  <description>
    <![CDATA[ Tapestry CRUD Example from Sandcast Software (http://sandcastsoftware.com/
    ]]></description>
  <property name="org.apache.tapestry.global-class"
    value="com.sandcast.examples.tapestry.Global"/>
  <library id="contrib"
    specification-path="/org/apache/tapestry/contrib/Contrib.library"/>
</application>
```

Alright, what does all of this mean?

- **application** - this specifies what Engine class you want to use (Tapestry allows for extending this and providing your own if you need to, we don't for this tutorial)
- **property** - this specifies an application property, in this case Global
- **page** - while not required you can specify pages here, in our case we are using it to alias a page that we will create (EditDVD)
- **library** - this specifies where Tapestry can find a library (a suite of additional or custom components) and what to call it when referencing it from inside of a Tapestry page or HTML file

Not too bad so far, it will get more complex (but not much more) as we progress through the tutorial series.

Alright, now that we've gone over the configuration now we need to finish up the rest of the application.

Add DVD/Edit DVD

The way that we have created our application these two pages are almost identical, with some slight differences. For our purposes we have combined the two pages together, but this may not always be advisable it will depend on your

requirements. Again, let's start with the HTML file.

EditDVD.html

```

<span jwcid="@Border">
  <form jwcid="@Form" delegate="ognl:beans.delegate">
    <span jwcid="@Hidden" value="ognl:dvdId"/>
    <span jwcid="@Hidden" value="ognl:newDvd"/>
    <table border="0" width="100%">
      <span jwcid="@Conditional" condition="ognl:beans.delegate.hasErrors">
        <tr>
          <td colspan="2">
            <span jwcid="@Delegator"
              delegate="ognl:beans.delegate.firstError">Error Message</span></td>
          </tr>
        </span>
        <tr>
          <td width="20%">
            <span jwcid="@FieldLabel"
              field="ognl:components.titleField">Title</span>
          </td>
          <td>
            <input jwcid="titleField" type="text" size="15"/>
          </td>
        </tr>
        <tr>
          <td width="20%">
            <span jwcid="@FieldLabel"
              field="ognl:components.costField">Short Name</span>
          </td>
          <td>
            <input jwcid="costField" type="text" size="15"/>
          </td>
        </tr>
        <span jwcid="@Conditional"
          condition="ognl:! newDvd">
          <tr>
            <td colspan="2" align="center">
              <input jwcid="@Submit" value="Update DVD"
                type="Submit" listener="ognl:listeners.updateDVD"/></td>
            </tr>
          </span>
          <span jwcid="@Conditional"
            condition="ognl:newDvd">
            <tr>
              <td colspan="2" align="center">
                <input jwcid="@Submit" value="New DVD" type="Submit"
                  listener="ognl:listeners.newDVD"/>
              </td>
            </tr>
          </span>
        </table>
      </form>
    </span>

```

This has quite a bit more going on than the listing page did, so let's break it down.

```

<span jwcid="@Border">

```

```

<form jwcid="@Form" delegate="ognl:beans.delegate">
<span jwcid="@Hidden" value="ognl:dvdId"/>
<span jwcid="@Hidden" value="ognl:newDvd"/>
<table border="0" width="100%">
<span jwcid="@Conditional" condition="ognl:beans.delegate.hasErrors">
  <tr>
    <td colspan="2"><span jwcid="@Delegator"
      delegate="ognl:beans.delegate.firstError">Error Message</span>
    </td>
  </tr>
</span>
</table>

```

First we have our Border component as before (remember it's a wrapper) then we have a Form component and a delegate (we'll get to it shortly, the delegate helps with form field validation). Inside the form we have a Hidden field for our dvd ID, which needs to bind to dvdId instead of something like Id, which will be explained when we get to the Java class file. Then we have another Hidden field for our newDvd boolean value, if we don't store it here then Tapestry loses the value of the property.

Next we have a Conditional component, this allows us to pass in an expression that when it validates to true will execute what is inside of it, in our case outputting the first form field validation error. What's next?

```

<tr>
  <td width="20%">
    <span jwcid="@FieldLabel" field="ognl:components.titleField">Title</span>
  </td>
  <td>
    <input jwcid="titleField" type="text" size="15"/>
  </td>
</tr>

```

We have the Title and Cost form fields. Now you will notice (when we get to the page specification) that these are tied to components.titleField instead of something like title, the reason for this is that we are using a special component called ValidField. ValidField allows us to do some fancy tricks with form field labels when the fields contain invalid data. This is also why the label and the field are bound to the same value as we are letting the component control the color of the label as well as adding in some extra text around the field (red asterisks).

Here is what our form looks like when validation fails:

You must enter a value for Entry Title.

Entry Title **

DVD Price

```

<span jwcid="@Conditional" condition="ognl:! newDvd">
  <tr>
    <td colspan="2" align="center"><input jwcid="@Submit" value="Update DVD" type="Submit" />
  </tr>
</span>
<span jwcid="@Conditional" condition="ognl:newDvd">
  <tr>
    <td colspan="2" align="center"><input jwcid="@Submit" value="New DVD" type="Submit" />
  </tr>
</span>

```

Finally we have the Submit button, or rather buttons. In this case we have two buttons, but depending on whether this is a new DVD or one that we are editing, only one of them shows up. Additionally these buttons are tied to specific actions in the Java class that we will look at shortly.

The EditDVD.page file

```

<page-specification class="com.sandcast.examples.tapestry.pages.EditDVD">
  <property-specification name="dvd" type="com.sandcast.examples.tapestry.crud.bo.DVD" />
  <bean name="delegate" class="org.apache.tapestry.valid.ValidationDelegate" />
  <property-specification name="callback" type="org.apache.tapestry.callback.ICallback" pers="true" />
  <property-specification name="newDvd" type="boolean" />

  <property-specification name="dvdId" type="int" />
  <property-specification name="title" type="java.lang.String" />
  <property-specification name="cost" type="float" />

  <bean name="reqValidator" class="org.apache.tapestry.valid.StringValidator">
    <set-property name="required" expression="true" />
  </bean>
  <component id="titleField" type="ValidField">
    <binding name="value" expression="title" />
    <binding name="validator" expression="beans.reqValidator" />
    <static-binding name="displayName" value="Entry Title" />
  </component>
  <component id="costField" type="ValidField">
    <binding name="value" expression="cost" />
    <binding name="validator" expression="beans.reqValidator" />
    <static-binding name="displayName" value="DVD Price" />
  </component>
</page-specification>

```

Again we have the root `page-specification` which tells Tapestry what Java class to link this to. We also have our current DVD setup as a property here, which is also defined in the Java class. Next we have a few new properties which we will now go over:

- **delegate** - This is our validation delegate that manages form input validation. You can specify your own delegate for specific look and feel requirements.
- **callback** - This allows us to specify what page called ours and then return to it (if specified).
- **reqValidator** - This is one of our validators, in this case a required validator. We can specify any number of validators or a custom validator should we choose.

- **titleField and costField** - These two fields use the ValidField component along with the validator that we just defined to make our components validatable.

We won't go into much detail on validation in this tutorial, but will cover it in more detail in the next two tutorials as we improve our application.

EditDVD.java (pt. 1)

```
public abstract class EditDVD extends BasePage implements PageRenderListener {
    private DVD dvd;
    private DvdDAOSimple dvdDAO;

    public abstract void setDvd(DVD dvd);
    public abstract DVD getDvd();

    public abstract void setNewDvd(boolean val);
    public abstract boolean getNewDvd();

    public abstract void setDvdId(int id);
    public abstract int getDvdId();

    public abstract void setTitle(String title);
    public abstract String getTitle();

    public abstract void setCost(float cost);
    public abstract float getCost();

    public abstract void setCallback(ICallback value);
    public abstract ICallback getCallback();

    protected IValidationDelegate getValidationDelegate() {
        return (IValidationDelegate) getBeans().getBean("delegate");
    }
}
```

Again, this is an Abstract class due to the Tapestry management of some of our properties. And we also have provide access to the delegate that we defined in the page specification as well as the callback property.

EditDVD.java (pt. 2)

```
public void pageBeginRender(PageEvent event) {
    if(!getNewDvd() && !event.getRequestCycle().isRewinding()) {
        setDvdId(getDvd().getId());
        setTitle(getDvd().getTitle());
        setCost(getDvd().getCost());
    }
}
```

While this is further down in the file it is probably the next most important piece. This method `pageBeginRender` is part of the `PageRenderListener` interface that we implemented above and gets called when the page is about to render, both on the initial call and on the submit. What we are doing here is checking to see whether we are creating a new DVD or if we are editing an existing DVD, because if we are editing an existing DVD then we need to grab the DVD that was set from the calling page and set some properties. Also we are checking to see what phase we are in, if the form or page is rewinding (i.e. - in the midst of being posted), then we need to ignore it, otherwise we'll get an error.

EditDVD.java (pt. 3)

```
public void newDVD(IRequestCycle cycle) {
    IValidationDelegate delegate = getValidationDelegate();
    if(delegate.getHasErrors())
        return;

    Global global = (Global) getEngine().getGlobal();
    DvdDAO dvdDAO = global.getDvdDAO();

    DVD dvd = new DVD();
    dvd.setCost(getCost());
    dvd.setTitle(getTitle());
    dvd.setId(-1);
    dvdDAO.saveDVD(dvd);

    ICallback callback = getCallback();

    if(callback == null)
        cycle.activate("Home");
    else
        callback.performCallback(cycle);
}
```

Since, `updateDVD` is almost identical to `newDVD` we'll just review `newDVD`. First, we are grabbing the delegate using the method we defined above. Then, we are checking to see if there are any form errors and returning if there are, this is a nice piece of functionality here, in case you need to do any additional error checking or business rule checking you can do it here before returning. Next we are creating a new DVD and then populating it with values from the form and then saving it using the DAO we created.

One obvious question at this point is why not store the DVD and access it through the properties? If this were a session, Tapestry would be storing that object in the session itself, which depending on the object could get quite hefty. We extract the values from it we need and then store those values as properties.

Then, why not store the id as `getId` and `setId`?

This one isn't so obvious, Tapestry has defined a number of properties in the `BasePage` class which `EditDVD` extends, overriding these will cause runtime

errors. Two of those fields are `id` and `name`, so we need to take caution when doing that.

One final thing to note about all pages in Tapestry, each page implementation requires the following be defined:

- **.html file** - This defines the layout of the page and what components you will use on the page.
- **.page file** - This file binds the Java and HTML files together.
- **.java file** - This provides the control logic specific to Tapestry. It should not be used to implement business logic, but rather call classes and helpers that do implement the business logic.

While these are required the Java file can be defined as the `org.apache.tapestry.html.BasePage` class if you do not need any special methods. As you can see this probably won't happen too often, but again it depends on your specific application requirements.

The Border component

This is the final piece to our Tapestry puzzle and quite possibly one of the simplest as well. Because this is a new component and not a page to create it we choose `New > Other > Tapestry > Tapestry Component` and are presented with this:

Create a new Tapestry Component file resource.

Component Name:

Project:

Namespace: Application Namespace

File Generation:

.jwc template:

Generate an associated HTML File?

.html template:

Choose a class for the specification on the next page...

Choosing the defaults and typing in the name "Border" we come to the next page

Create a new Tapestry Page file resource.

Page Name:

Project:

Namespace:

File Generation:

.page template:

Generate an associated HTML File?

.html template:

Choose a class for the specification on the next page...

Again, we are going to create our own class to go along with it. Although we aren't going to do anything special with it for this tutorial. Let's place this new class in `com.sandcast.examples.tapestry.components`. Below is the component file that we want to create:

```
<component-specification class="com.sandcast.examples.tapestry.components.Border" allow-body="y"
</component-specification>
```

And the HTML file

```
<html jwcid="@Shell" title="Tapestry CRUD Listing example - Sandcast Software">
  <body jwcid="@Body">
    <table border="0" width="100%">
      <tr>
```

```
<td width="20%" valign="top">
  <span jwcid="@PageLink" page="Home">Show DVDs</span><br />
</td>
<td valign="top"><span jwcid="@RenderBody">Page content goes here.</span></td>
</tr>
</table>
</body>
</html>
```

We now have some special components (and new attribute values) that we should examine:

- **Shell** - This provides us with access to the HTML headers and wraps the body element.
- **Body** - This creates a body element inside of our template
- **RenderBody** - This allows us to render the content in between the tags that contain this component. Everything in between the `` tag here gets rendered. You can only have one `RenderBody` on a page. If you have more complex needs you should look at the `RenderBlock` component.

Why do we need all these special components just to create an HTML page?

Tapestry needs access to much of the page that gets rendered for several reasons:

- **Custom JavaScript** - There are many instances when Tapestry needs to insert JavaScript into the head element, or into the form element
- **Stylesheets** - It's possible to dynamically change a stylesheet based on a user's preference

While it isn't absolutely required to hand over template control to Tapestry in most cases it is desirable to do so.

Section 5. Wrap-up

What we learned

Tools

- **Spindle** - We've shown the basics of using Spindle to generate our Tapestry application
- **Eclipse** - We've shown how to install the Spindle plugin to Eclipse 3.0
- **Maven** - We've shown (briefly) how to use Maven and Eclipse together

Techniques

- **DAO Pattern** - How to implement a DAO Pattern with Tapestry
- **Tapestry application** - How to create a real-world Tapestry application
- **Component creation** - How to create a custom component (Border) and use it inside of our Tapestry application

Topics

- **CRUD Listing** - What is a CRUD listing?
- **Component vs. MVC architecture** - What is a component-based architecture as compared to an MVC-based architecture

Section 6. Feedback and Resources

Feedback

Thank you for downloading this tutorial if you have any questions or feedback please email brownbag@sandcastsoftware.com

Resources

Tapestry

- Core - The core Tapestry components can be found at <http://jakarta.apache.org/tapestry>
- Spindle Eclipse Plugin - The latest Spindle plugin can be found at <http://spindle.sourceforge.net/unstable/> (note this link is for use inside of Eclipse)
- Tapestry Wiki - A community area for sharing tips, code, and other resources is at: <http://wiki.apache.org/jakarta-tapestry/>
- Tapestry Deli - A place for components and tools not suited for the contrib area, <http://t-deli.com/>
- Tacos - Tapestry components, a place for components to mature before being placed in contrib, <http://sourceforge.net/projects/tacos/>
- Howard's Blog (the creator of Tapestry) - <http://howardlewisship.com/blog>
- Javassist bytecode library - <http://www.jboss.org/developers/projects/javassist.html>
- OGNL - <http://ognl.org>

Eclipse can be downloaded from <http://eclipse.org> - this tutorial used Eclipse 3.0.

You can learn more about Maven at <http://maven.apache.org>.

About the author

Warner Onstine is the President and Founder of Sandcast Software, a company focused on providing pre-built and custom solutions to small and middle-sized companies and organizations. Their specific focus is on communities and teams, by providing Caelum, their software for community publishing, and COAL (in conjunction with Kate Rhodes of Masukomi.org) their software for managing small, medium and large software and design projects. Additionally they provide consulting to Libraries and Library resource providers, to assist in implementing new solutions to existing problems, helping to bring the Library

closer to their customer and the provider closer to the Library.

He is also a published author of the book "*Professional Java Tools for Extreme Programming*" published under Wrox, of which he wrote 5 chapters, including "Managing Projects with Maven".

Special Thanks

Special Thanks go out to [Howard L. Ship](http://howardlewisship.com/) (the creator of Tapestry), Erik Hatcher ([ehatcher solutions](http://ehatchersolutions.com/)), [personal blog](http://blogscene.org/erik)), [Danny Mandel](http://tolweb.org), and [Joe Andolina](http://www.andomation.com) for Code and Tutorial Review. Because of them this is better than it was ;-). Thanks guys!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/.