

# 12

## General Wicket Patterns

In this chapter, we will cover:

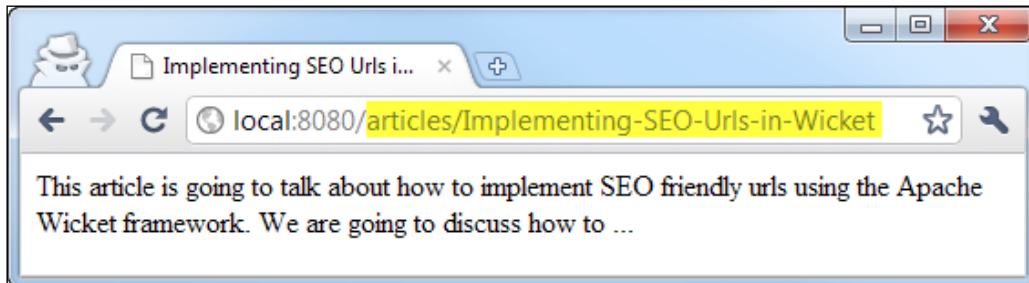
- ▶ How to support SEO-friendly URL paths
- ▶ Building component hierarchy lazily
- ▶ Storing additional data with components, sessions, and application
- ▶ Creating components that use more than one model
- ▶ Contributing unminified JavaScript resources at development time to make debugging easier
- ▶ Managing page title
- ▶ Creating a page-based main menu

### Introduction

In this chapter, we are going to look at a wide variety of topics. We are going to look at some of the general issues that come up with implementing individual components or interactions between multiple components. While the recipes in this chapter are useful by themselves, they also demonstrate patterns for implementing usecases and solving problems that come up over and over again.

## How to support SEO-friendly URL paths

Behaving in an SEO-friendly manner is a must-have feature for most web applications. An important part of the SEO strategy is to have clear and readable URLs for parts of the application that can be accessed directly by users and search engines alike. Wicket supports this feature by allowing the user to "mount" pages onto URL prefixes; in this chapter we will examine how to implement this in an application that displays articles much like a blog:



### Getting ready

Let's get started by building the domain model for our demo application.

Build the `Article` object:

`Article.java`

```
public class Article {  
    public String title, key, content;  
}
```

Build an object that acts as a store or a repository for the articles:

`ArticleStore.java`

```
public class ArticleStore {  
    private List<Article> articles = new ArrayList<Article>();  
    // sample code provides methods for adding, getting, and retrieving  
    articles from the articles collection  
}
```

Create a singleton instance of the store and make it accessible to Wicket components:

WicketApplication.java

```
public class WicketApplication extends WebApplication {  
    private ArticleStore store = new ArticleStore();  
    public ArticleStore getStore() {  
        return store;  
    }  
}
```

Create a Wicket model that can retrieve an article given its key:

ArticleModel.java

```
public class ArticleModel extends  
    LoadableDetachableModel<Article>{  
    private String key;  
    public ArticleModel(Article article) {  
        super(article);  
        this.key = article.key;  
    }  
    public ArticleModel(String key) {  
        this.key = key;  
    }  
    protected Article load() {  
        return  
            ((WicketApplication)Application.get())  
                .getStore()  
                .getByKey(key);  
    }  
}
```

Create a page that will display an article:

ArticlePage.java

```
// for markup refer to ArticlePage.html in the code bundle  
public class ArticlePage extends WebPage {  
    public ArticlePage(IModel<Article> article) {  
        init(article);  
    }  
    protected void init(IModel<Article> article) {
```

```
        add(new Label("title",
            new PropertyModel(article, "title")));
        add(new Label("content",
            new PropertyModel(article, "content")));
    }
}
```

## How to do it...

1. Mount the page in Wicket's application class:

```
WicketApplication.java
public class WicketApplication extends WebApplication {
    protected void init() {
        mount(
            new IndexedParamUrlCodingStrategy(
                "/articles", ArticlePage.class));
    }
}
```

2. Modify ArticlePage to work with the mount:

```
public class ArticlePage extends WebPage {

    public ArticlePage(PageParameters params) {
        final String key = params.getString("0");
        IModel<Article> article = new ArticleModel(key);
        init(article);
    }
}
```

3. Add a method to generate a URL based on the mount to the ArticlePage:

```
public class ArticlePage extends WebPage {

    public static CharSequence urlFor(Article article) {
        RequestCycle rc = RequestCycle.get();
        PageParameters params = new PageParameters();
        params.put("0", article.key);
        return rc.urlFor(ArticlePage.class, params);
    }
}
```

4. Create a link component that makes it easy to link to the `ArticlePage`:

```
public class ArticleLink extends Link<Article> {
    public ArticleLink(String id, IModel<Article> model) {
        super(id, model);
    }
    public final void onClick() {
        // will never be called since we
        // generate a bookmarkable url
    }
    protected CharSequence getURL() {
        return ArticlePage.urlFor(getModelObject());
    }
}
```

5. Add an article link to the first article in the home page:

```
HomePage.java
// for markup refer to HomePage.html in the code bundle
public class HomePage extends WebPage {
    public HomePage(final PageParameters parameters) {
        Article article =
            ((WicketApplication) getApplication())
                .getStore().get(0);
        add(new ArticleLink("link", new ArticleModel(article)));
    }
}
```

## How it works...

In this example we wish the articles to be accessible by URLs in the following format:

```
/articles/<articleKey>
```

As the `articleKey` variable is encoded without a parameter name we are going to use Wicket's `IndexedParamUrlCodingStrategy` which encodes parameters by their index instead of their name:

WicketApplication.java

```
protected void init() {
    mount(
        new IndexedParamUrlCodingStrategy(
            "/articles", ArticlePage.class));
}
```

Now let's implement the code that can build a URL to an article. As the URL links to the `ArticlePage` we will put the code there. The first thing we must do is construct the `PageParameters` object which is used to pass parameters which should appear on the URL. As we are using `IndexedParamUrlCodingStrategy` we are going to specify the name of the `articleKey` parameter as "0":

`ArticlePage.java`

```
public static CharSequence urlFor(Article article) {
    PageParameters params = new PageParameters();
    params.put("0", article.key);
}
```

We are now ready to construct the URL which we do using `RequestCycle`'s `urlFor()` method:

`ArticlePage.java`

```
public static CharSequence urlFor(Article article) {
    PageParameters params = new PageParameters();
    params.put("0", article.key);
    RequestCycle rc = RequestCycle.get();
    return rc.urlFor(ArticlePage.class, params);
}
```

Now that we can encode an article into URL we must also be able to decode it – we do this in `ArticlePage`'s constructor. As we want to access parameters passed on URL we create a constructor that accepts a `PageParameters` object and perform the reverse of what we've done in the `urlFor(Article)` method we have created:

`ArticlePage.java`

```
public ArticlePage(PageParameters params) {
    final String key = params.getString("0");
    IModel<Article> article = new ArticleModel(key);
    init(article);
}
```

We now have an SEO-friendly way to generate URLs to our articles.

## There's more...

In the next section, we will examine how to make the application that uses book-markable URLs a bit more resilient.

## Responding to bad URLs with a 404

One of the downsides of using bookmarkable URLs is that users can mess around with them to try and break the application or they may become obsolete over time. Let's implement the code that makes sure the `articleKey` passed on the URL is valid and if it is not return an HTTP 404 error:

`ArticlePage.java`

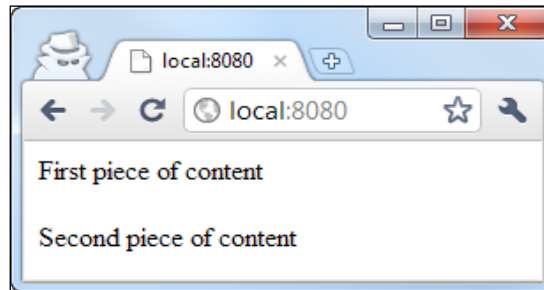
```
public ArticlePage(PageParameters params) {
    final String key = params.getString("0");
    IModel<Article> article = new ArticleModel(key);
    if (article.getObject() == null) {
        throw new AbortWithWebErrorCodeException(404);
    }
    init(article);
}
```

The check is rather simple: it attempts to retrieve the article by the key and if it is not found it throws an exception which will result in the 404 error code returned to the browser.

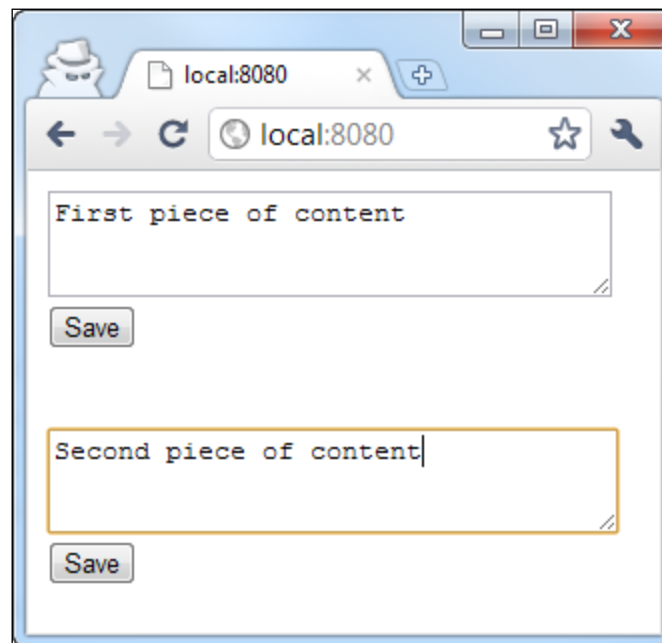
## Building component hierarchy lazily

Most of the time the component hierarchy in Wicket pages is constructed inside component constructors – this works well for a large majority of usecases. However, there are times when we want to delay construction of the hierarchy until we have access to the `Page` object, and in this recipe we are going to see how to do that. Suppose we are building an application with CMS-like functionality where pages can be constructed in either a `VIEW` or a `EDIT` mode. Some components may wish to react to this page mode by displaying different representations of them. For example, a component used to manage text content on the page can display itself as text when the page is in `VIEW` mode and as an editable text area when the page is in `EDIT` mode:

A page with two `ContentPanel` components in VIEW mode:



A page with two `ContentPanel` components in EDIT mode:



## Getting ready

Let's get started by building the home page.

Build the base page for all pages that support VIEW and EDIT modes:

`BasePage.java`

```
public abstract class BasePage extends WebPage {  
    public static enum Mode { VIEW, EDIT }  
}
```

```

    private Mode mode;

    // getter/setter for mode
}

```

Implement the home page:

HomePage.java

```

// for markup refer to HomePage.html in the code bundle
public class HomePage extends BasePage {
    public HomePage() {
        setMode(Mode.VIEW);
        add(
            new ContentPanel("content1",
                Model.of("The first piece of content")));
        add(
            new ContentPanel("content2",
                Model.of("The second piece of content")));
    }
}

```

## How to do it...

1. Implement the ContentPanel and make it aware of the page's mode:

ContentPanel.html

```
<wicket:panel><div wicket:id="content"></div></wicket:panel>
```

ContentPanel\$EditPanel.html



Note that because EditPanel is an inner class of ContentPanel its full class name is ContentPanel\$EditPanel and this is why the markup file is named: ContentPanel\$EditPanel.html.

```

<wicket:panel>
<form wicket:id="form">
    <textarea wicket:id="content"></textarea><br/>
    <input type="submit" value="Save"/>
</form>
</wicket:panel>

```

```
ContentPanel.java
public class ContentPanel extends Panel {
    public ContentPanel(String id, IModel<String> content) {
        super(id, content);
    }

    protected void onInitialize() {
        super.onInitialize();

        Mode mode = Mode.VIEW;
        if (getPage() instanceof BasePage) {
            mode = ((BasePage) getPage()).getMode();
        }

        switch (mode) {
            case EDIT:
                add(new EditPanel("content"));
                break;
            case VIEW:
                add(new Label("content", getDefaultModel()));
                break;
        }
    }

    public class EditPanel extends Panel {
        public EditPanel(String id) {
            super(id);

            Form<?> form = new Form<Void>("form");
            add(form);
            form.add(new TextArea("content", ContentPanel.this
                .getDefaultModel()));
        }
    }
}
```

### How it works...

Instead of creating `ContentPanel`'s component hierarchy in its constructor, we create it inside the `onInitialize()` method. This method is called on components when a path exists from the page to the component (when `getPage()` will return the page instance to which the component is ultimately added). This method is also guaranteed to only be called once on the component so there is no need to check if component hierarchy is already created.

We begin by implementing the method and calling super; super call is required in this method and Wicket will throw an exception if it is not called:

ContentPanel.java

```
protected void onInitialize() {
    super.onInitialize();
}
```

Then we figure out the mode, defaulting to VIEW if we are in a page that does not support page modes:

ContentPanel.java

```
protected void onInitialize() {
    super.onInitialize();

    Mode mode = Mode.VIEW;
    if (getPage() instanceof BasePage) {
        mode = ((BasePage) getPage()).getMode();
    }
}
```

Now we are ready to construct the class hierarchy of the panel base on the mode:

ContentPanel.java

```
protected void onInitialize() {
    super.onInitialize();

    Mode mode = Mode.VIEW;
    if (getPage() instanceof BasePage) {
        mode = ((BasePage) getPage()).getMode();
    }

    switch (mode) {
        case EDIT:
            add(new EditPanel("content"));
            break;
        case VIEW:
            add(new Label("content", getDefaultModel()));
            break;
    }
}
```

## There's more...

In the next section, we will see how to make our component a little bit more dynamic.

### Responding to changes on the fly

In the preceding example, while the `ContentPanel` supports page modes, it does not support switching between them dynamically. Let's see how we can modify it so it supports page mode switches.

This time, instead of using `onInitialize()` method which is only called once we are going to use `onBeforeRender()` which is called before every component render:

`ContentPanel.java`

```
protected void onBeforeRender() {
    super.onBeforeRender();
}
```

Now add back the code that figures out which mode we are in:

`ContentPanel.java`

```
protected void onBeforeRender() {
    Mode mode = Mode.VIEW;
    if (getPage() instanceof BasePage) {
        mode = ((BasePage) getPage()).getMode();
    }
    super.onBeforeRender();
}
```

As `onBeforeRender()` is called multiple times we do not want to keep rebuilding the child component hierarchy unless the mode has changed; let's implement this by remembering the mode for which we have created the hierarchy previously:

`ContentPanel.java`

```
public class ContentPanel extends Panel {
    private Mode lastMode;

    protected void onBeforeRender() {
        Mode mode = Mode.VIEW;
        if (getPage() instanceof BasePage) {
            mode = ((BasePage) getPage()).getMode();
        }
        if (lastMode != mode) {
            this.lastMode = mode;
        }
    }
}
```

```

        super.onBeforeRender();
    }
}

```

And with that we are ready to build the child hierarchy:

ContentPanel.java

```

protected void onBeforeRender() {
    Mode mode = Mode.VIEW;
    if (getPage() instanceof BasePage) {
        mode = ((BasePage) getPage()).getMode();
    }
    if (lastMode != mode) {
        switch (mode) {
            case EDIT:
                addOrReplace(new EditPanel("content"));
                break;
            case VIEW:
                addOrReplace(
                    new Label("content", getDefaultModel()));
                break;
        }
        this.lastMode = mode;
    }
    super.onBeforeRender();
}

```



Note that we use `addOrReplace(Component)` method instead of `add(Component)` because a component with the specified id may already exist and if we use `add(Component)` Wicket will throw an exception, whereas if we use `addOrReplace(Component)` Wicket will replace the old component with the new one.

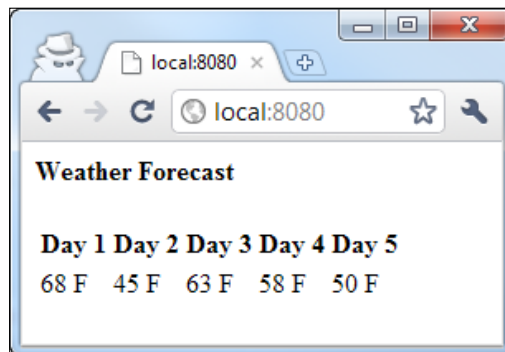
You may have noticed that when using `onInitialize()` the first thing we did was call `super`, whereas when using `onBeforeRender()` we called `super` last. This is because the `super` call to `onBeforeRender()` continues the cascade of the call to the child component hierarchy and since we are potentially adding new components and want `onBeforeRender()` to be called on them we have to call `super` after we have added them.

As Wicket forces the initialization order when using `onInitialize()`, the `super` call does not affect the cascade; however, because this is an initialization method we should call `super` first to allow the superclass to initialize itself first – just like Java forces the `super` to be the first thing called in constructors.

## Storing additional data with components, sessions, and application

Wicket strives to be as type-safe as possible, for this reason users are encouraged to store data in context classes such as the `Application`, `Session`, and `RequestCycle` as fields in a custom subclass and provide proper getters and setters for those fields. This mostly works great; however, there are times when subclassing one of these classes is not convenient. For example, because Java does not support multiple inheritance component libraries are not encouraged to provide custom `Session` or `RequestCycle` subclasses because the user would not be able to use two libraries at the same time since they cannot extend from two `Session` subclasses. To work around these cases Wicket provides a way to attach arbitrary data to `Application`, `Session`, `RequestCycle`, and `Component` subclasses called the `MetaData` storage. In this recipe, we will see how to use this storage to cache some data on session level.

We are going to build a weather widget that will display a five day forecast. It will also cache the forecast in the `Session` so no matter how many times it is rendered the forecast, which is presumably expensive to retrieve, is only retrieved once per user session. As we are going to store the forecast in `Session`'s metadata, we do not need to supply a `Session` subclass the user would have to use in order to enable the caching.



### Getting ready

Let's get started by implementing the weather widget without caching.

Build the domain and the logic to "retrieve" the forecast:

`WeatherForecast.java`

```
public class WeatherForecast implements Serializable {  
    public int[] temperature;
```

```

    public void retrieve() {
        // sample code fills in the temperature array with random values
    }
}

```

Build the widget:

WeatherForecastWidget.java

```

// for markup refer to WeatherForecastWidget.html in the code bundle
public class WeatherForecastWidget extends Panel {
    public WeatherForecastWidget(String id) {
        super(id);
        add(new Label("t1",
            new PropertyModel(this,
                "localForecast.temperature.0")));
        // sample code adds four more labels for elements 2-5 of the
        array
    }

    public WeatherForecast getLocalForecast() {
        WeatherForecast forecast = new WeatherForecast();
        forecast.retrieve();
        return forecast;
    }
}

```

Drop the widget into a page. Refer to HomePage.html and HomePage.java in the code bundle for a sample page that hosts the widget.

## How to do it...

1. Modify the widget to enable caching of forecast data in Session:

```

WeatherForecastWidget.java
public class WeatherForecastWidget extends Panel {
    private static final MetadataKey<WeatherForecast> KEY =
        new MetadataKey<WeatherForecast>() {};

    public WeatherForecast getLocalForecast() {
        WeatherForecast forecast =
            getSession().getMetadata(KEY);
        if (forecast == null) {
            forecast = new WeatherForecast();
            forecast.retrieve();
            getSession().setMetadata(KEY, forecast);
            getSession().bind();
        }
    }
}

```

```
    }  
    return forecast;  
  }  
}
```



Note that after putting the forecast into `Session`'s metadata we also call `Session`'s `bind()` method – this will force Wicket to store the `Session` object in HTTP session. If we omit this call there is a chance Wicket will not bind the `Session` for us and on the next request the forecast data will be retrieved again.

## How it works...

The metadata store is accessed via two methods:

```
public final <M extends Serializable> M getMetaData(  
    final MetaDataKey<M> key)  
  
public final void setMetaData(  
    final MetaDataKey<M> key, final M object)
```

It works much like a `Map`, but the keys are all subclasses of `MetaDataKey` object. Since the key subclasses carry the type of object they store the metadata store is more type-safe than just a `Map<Object, Object>` because the users can only retrieve the type of object they stored.

In order to store the forecast we first need a metadata key. The keys are usually implemented as private static anonymous subclasses of `MetaDataKey` base class:

WeatherForecastWidget.java

```
public class WeatherForecastWidget extends Panel {  
    private static final MetaDataKey<WeatherForecast> KEY =  
        new MetaDataKey<WeatherForecast>() {};  
}
```

Now that we have the key we can store and retrieve instances of `WeatherForecast` from metadata:

WeatherForecastWidget.java

```
public WeatherForecast getLocalForecast() {  
    WeatherForecast forecast =  
        getSession().getMetaData(KEY);  
    if (forecast == null) {  
        forecast = new WeatherForecast();  
    }  
}
```

```

        forecast.retrieve();
        getSession().setMetaData(KEY, forecast);
        getSession().bind();
    }
    return forecast;
}

```

### More Info Section 1

In the previous recipe, we cached the data in Session, but we can just as easily cache the data in Application:

WeatherForecastWidget.java

```

public WeatherForecast getLocalForecast() {
    WeatherForecast forecast =
        getApplication().getMetaData(KEY);
    if (forecast == null) {
        forecast = new WeatherForecast();
        forecast.retrieve();
        getApplication().setMetaData(KEY, forecast);
    }
    return forecast;
}

```

Or RequestCycle:

WeatherForecastWidget.java

```

public WeatherForecast getLocalForecast() {
    WeatherForecast forecast =
        getRequestCycle().getMetaData(KEY);
    if (forecast == null) {
        forecast = new WeatherForecast();
        forecast.retrieve();
        getRequestCycle().setMetaData(KEY, forecast);
        getSession().bind();
    }
    return forecast;
}

```



Note that access to Application and Session objects is not synchronized so some care must be taken if concurrent operations must be taken into account.

## Creating components that use more than one model

Sometimes components require two models to function. While all Wicket components come with a single model slot built-in it is easy to support more than one. In this recipe, we are going to build a simple `StyledLabel` component which uses one model slot to display text, while using a second slot to append a CSS style to its tag.

### Getting ready

Implement a model that will return a random style string:

`RandomStyleModel.java`

```
public class RandomStyleModel extends
    LoadableDetachableModel<String> {
    protected String load() {
        String[] styles = new String[] {
            "red", "green", "blue" };
        return styles[(int) (Math.random() * 3)];
    }
}
```

Prepare a page to host a few instances of our `StyledLabel`:

`HomePage.java`

```
// for markup refer to HomePage.html in the code bundle
public class HomePage extends WebPage {
    public HomePage(final PageParameters parameters) {
        add(new StyledLabel("label1",
            Model.of("Label 1"), new RandomStyleModel()));
        // sample code adds two more instances of StyledLabel
    }
}
```

### How to do it...

1. Build `StyledLabel` component:

```
public class StyledLabel extends Label {
    private IModel<String> style;

    public StyledLabel(String id, IModel<?> model, IModel<String>
style) {
```

```

        super(id, model);
        this.style = style;
    }

    protected void onComponentTag(ComponentTag tag) {
        super.onComponentTag(tag);
        tag.put("class", style.getObject());
    }

    protected void onDetach() {
        style.detach();
        super.onDetach();
    }
}

```

### How it works...

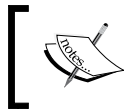
We are free to store model instances as fields in our components if we need more than the default model; however, we must take care to detach the models ourselves since Wicket does not know about any model other than the one stored in the default slot. In order to detach models we use Component's `onDetach()` method:

StyledLabel.java

```

    protected void onDetach() {
        style.detach();
        super.onDetach();
    }

```



Note that it is important to call `super` inside `onDetach()` overrides so superclasses are given a chance to detach their state.

### More Info Section 1

There is a small subclass of models, namely ones that implement `IComponentAssignedModel` interface that wish to be notified when they are bound to a component. In the recipe above if such a model is passed to the style model of our `StyledLabel` it will not work correctly because we do not support the `IComponentAssignedModel` contract by notifying the model it is being bound to our `StyledLabel`. We can easily fix this by changing our constructor like this:

StyledLabel.java

```

    public StyledLabel(String id, IModel<?> model,
        IModel<String> style) {

```

```
        super(id, model);
        this.style = wrap(style);
    }
```

The `wrap()` method will make sure `IComponentAssignedModel` contract is properly supported.

Note that if we had a setter for the style model it would have to be similarly changed:

StyledLabel.java

```
    public void setStyleModel(IModel<String> style) {
        this.style = wrap(style);
    }
```

## Contributing unminified JavaScript resources at development time to make debugging easier

Minifying JavaScript resources to users has become a best practice because it decreases application load times and improves user experience; serving minified JavaScript to developers who are working on the application, however, is highly undesirable. Minified JavaScript is difficult to work with and debug; developers need to work with unminified resources when working on the application. In this recipe, we will see how to serve minified JavaScript resources to users while serving unminified JavaScript resources to developers.

### Getting ready

In this recipe, we will use the popular jQuery JavaScript library.

Download both minified (`jquery-1.4.4.js`) and unminified (`jquery-1.4.4.dev.js`) versions of jQuery and save them next to the `ResourceReference` subclass we will create later to serve the minified version:

JQueryReference.java

```
public class JQueryReference extends ResourceReference {
    public JQueryReference() {
        super(JQueryReference.class, "jquery-1.4.4.js");
    }
}
```

Create a page with some simple JavaScript functionality:

HomePage.html

```
<html xmlns:wicket>
<body>
  <div id="container">
    <a href="#" id="link">Check jQuery</a>
  </div>
  <script>
    $("#link").click(function() {
      alert("jQuery is installed!");
    });
  </script>
</body>
</html>
```

HomePage.java

```
public class HomePage extends WebPage
    implements IHeaderContributor {

    public void renderHead(IHeaderResponse response) {
        response.renderJavascriptReference(
            new JQueryReference());
    }
}
```

## How to do it...

1. Modify JQueryReference to serve unminified version when application is in development mode:

```
JQueryReference.java
public class JQueryReference extends ResourceReference {
    public JQueryReference() {
        super(JQueryReference.class, script());
    }

    private static String script() {
        Application app = Application.get();
        if (Application.DEVELOPMENT.equals(
            app.getConfigurationType())) {
            return "jquery-1.4.4.dev.js";
        } else {
            return "jquery-1.4.4.js";
        }
    }
}
```

## How it works...

The JQueryReference checks whether or not the Wicket application is deployed in DEPLOYMENT or DEVELOPMENT mode:

```
private static String script() {
    Application app = Application.get();
    if (Application.DEVELOPMENT.equals(
        app.getConfigurationType())) {
        return "jquery-1.4.4.dev.js";
    } else {
        return "jquery-1.4.4.js";
    }
}
```

The reference returns the name of unminified resource if DEVELOPMENT mode is detected. Since the application is deployed in DEVELOPMENT mode when developers are working on it, the developers will see unminified versions of JavaScript which will make debugging problems easier.

## More Info Section 1

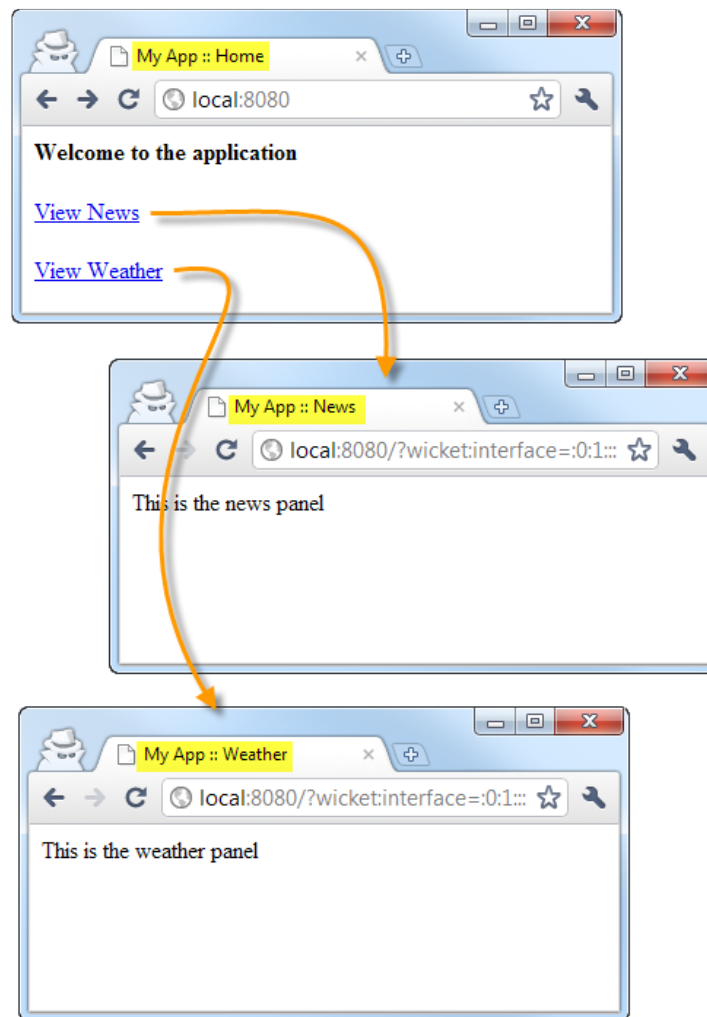
Wicket also ships with a JavascriptResourceReference, JavaScript resources served with this reference will be minified and gzipped by Wicket on the fly when Wicket is configured in deployment mode. This is a great alternative to storing two versions of JavaScript libraries. We can modify our recipe above to use this reference as follows:

JQueryReference.java

```
public class JQueryReference extends JavascriptResourceReference {
    public JQueryReference() {
        super(JQueryReference.class,
            "jquery-1.4.4.dev.js");
    }
}
```

## Managing page title

A common problem when using component-based framework is managing the page title. There can be many causes for this difficulty, including: page class hierarchies that were not designed with title management in mind, components other than paging needing to contribute to the title. In this recipe, we will see how to build a solution that solves the two problems above. In order to demo our solution we are going to build a simple panel-based page-flow:



## Getting ready

Let's get started by implementing the page-flow without title management.

Build the News panel:

NewsPanel.java

```
// for markup refer to NewsPanel.html in the code bundle
public class NewsPanel extends Panel {
    public NewsPanel(String id) { super(id); }
}
```

Build the Weather panel:

WeatherPanel.java

```
// for markup refer to WeatherPanel.html in the code bundle
public class WeatherPanel extends Panel {
    public WeatherPanel(String id) { super(id); }
}
```

Build the Home panel:

HomePanel.java

```
// for markup refer to HomePanel.html in the code bundle
public class HomePanel extends Panel {
    public HomePanel(String id) {
        super(id);
        add(new Link("news") {
            public void onClick() { onNews(); }
        });
        add(new Link("weather") {
            public void onClick() { onWeather(); }
        });
    }
    protected void onNews() {
        replaceWith(new NewsPanel(getId()));
    }
    protected void onWeather() {
        replaceWith(new WeatherPanel(getId()));
    }
}
```

Implement the container page:

HomePage.java

```
// for markup refer to HomePage.html in the code bundle
public class HomePage extends WebPage {
    public HomePage() {
        add(new Label("title",
            new PropertyModel(this, "title")));
        add(new HomePanel("content"));
    }
    public String getTitle() {
        return "";
    }
}
```

```
    }
}
```

## How to do it...

1. Implement the `TitleContributor` interface:


```
TitleContributor.java
public interface TitleContributor {
    String getTitleContribution();
}
```

2. Implement page title construction:

```
HomePage.java
public class HomePage extends WebPage
    implements TitleContributor {

    public String getTitle() {
        final StringBuilder title = new StringBuilder();
        title.append(getTitleContribution());
        visitChildren(new IVisitor<Component>() {
            public Object component(Component component) {
                if (component instanceof TitleContributor) {
                    title.append(" :: ");
                    title.append(((TitleContributor) component)
                        .getTitleContribution());
                }
                return CONTINUE_TRAVERSAL;
            }
        });
        return title.toString();
    }

    public String getTitleContribution() { return "My App"; }
}
```

 Notice that the visitor method `visitChildren()` does not visit the component itself and so we have to add the page's contributions manually: `title.append(getTitleContribution());`

3. Wire in title contribution to content panels:

```
HomePanel.java
public class HomePanel extends Panel
    implements TitleContributor {
    public String getTitleContribution() { return "Home"; }
}
```

```
}
NewsPanel.java
public class NewsPanel extends Panel
    implements TitleContributor {
    public String getTitleContribution() { return "News"; }
}
WeatherPanel.java
public class WeatherPanel extends Panel
    implements TitleContributor {
    public String getTitleContribution() { return "Weather"; }
}
```

## How it works...

The idea behind our system is simple: we are going to allow pages and components to contribute to the page title by implementing the `TitleContributor` interface. When the page is ready to render the title we are going to walk the component hierarchy of the page, looking for all `TitleContributors` using a visitor:

```
HomePage.java

public String getTitle() {
    final StringBuilder title = new StringBuilder();
    title.append(getTitleContribution());
    visitChildren(new IVisitor<Component>() {
        public Object component(Component component) {
            if (component instanceof TitleContributor) {
                title.append(" :: ");
                title.append(((TitleContributor) component)
                    .getTitleContribution());
            }
            return CONTINUE_TRAVERSAL;
        }
    });
    return title.toString();
}
```

And when we find a title contributor, we get its contribution and append it to the buffer which will ultimately become the final title.

The system is very flexible and works for page-based or panel-based navigation scenarios, or even for a mixture of both.

## More Info Section 1

In the recipe above the order of title contributions is defined by the depth of the component in the component hierarchy– components closer to the top (the page) have their contributions appended first, while components deepest in the hierarchy have their contributions appended last. This usually works for most applications because the component hierarchy of components that wish to contribute to the title is not large. However, it is not too difficult to add weights to title contributions – allowing contribution order that does not depend on the hierarchy. To do this we would modify `TitleContributor` to also provide a weight:

`TitleContributor.java`

```
public interface TitleContributor {
    String getTitleContribution();
    int getTitleContributionWeight();
}
```

We can then modify the code that combines contributions to take weight into account. To facilitate sorting let's create an object to hold contribution and its weight:

`HomePage.java`

```
private static class Contribution implements Comparable<Contribution>
{
    public String text;
    public int weight;

    public Contribution(TitleContributor c) {
        text = c.getTitleContribution();
        weight = c.getTitleContributionWeight();
    }

    public int compareTo(Contribution o) {
        return o.weight - weight;
    }
}
```

Now we can modify the `getTitle()` method to take weight into account:

`HomePage.java`

```
public String getTitle() {
    final List<Contribution> contributions =
        new ArrayList<Contribution>();
    contributions.add(new Contribution(this));
    visitChildren(new IVisitor<Component>() {
        public Object component(Component component) {
            if (component instanceof TitleContributor) {
                contributions.add(
```

```
        new Contribution(
            (TitleContributor) component));
    }
    return CONTINUE_TRAVERSAL;
}
});

Collections.sort(contributions);
StringBuilder title = new StringBuilder();
for (Contribution c : contributions) {
    if (title.length() > 0) {
        title.append(" :: ");
    }
    title.append(c.text);
}
return title.toString();
}
```

## Creating a page-based main menu

Most Wicket applications are structured around a set of book-markable pages that represent main entry points. There is also, usually, a menu on each of these pages that links to all other such pages – the main menu of the web application. In this recipe, we will build a component to model such main menu. As most such menus are represented in the markup as an unordered list this is what our menu component is going to render.

### Getting ready

Let's get started by creating a set of bookmarkable pages which will constitute main entries in our demo application.

Construct the base page for main-entry pages:

BasePage.html

```
<html>
  <body>
    <wicket:child/>
  </body>
</html>
```

BasePage.java

```
public abstract class BasePage extends WebPage
{
    public BasePage() {}
}
```

Create a couple of main-entry pages:

HomePage.html

```
<wicket:extend>This is our home page.</wicket:extend>
```

HomePage.java

```
public class HomePage extends BasePage {}
```

ProductsPage.html

```
<wicket:extend>
This page contains product listing so you can buy our awesome
products.
</wicket:extend>
```

ProductsPage.java

```
public class ProductsPage extends BasePage {}
```

AboutUsPage.html

```
<wicket:extend>
This page contains information about our store
</wicket:extend>
```

AboutUsPage.java

```
public class AboutUsPage extends BasePage {}
```

## How to do it...

Before we get started, I should note that this recipe gives us an opportunity to examine how to build components that render markup. While the need to render markup directly is seldom, after all - doing so with components is much easier, knowing how to do it is sometimes useful. There are a few reasons to do so:

1. Sometimes it is easier to render markup imperatively rather than representing it using a component hierarchy.
2. Writing out markup directly allows us to optimize the amount of state components take up in session.

While we start out with rendering a flat unordered list in the main recipe, in a later section we will see how to create hierarchical menus which require rendering of hierarchical lists. Rendering a hierarchical structure would require building a panel or a fragment that supports recursion; which is not trivial. Writing out markup directly allows us to accomplish this in a simpler manner.

If we were to use components to build the menu, we would need roughly 11 components to represent a three item menu: a component to represent the menu, a repeater, a repeater item for each menu item, a link for each menu item, and a label for each menu item. Writing out markup directly allows us to use a single component to render unlimited numbers of menu items. As one component takes up less state in session when compared to 11 components this approach can be seen as an optimization technique for rare applications that require the absolute minimal use of stateful resources.



A word of warning: this approach is by no means recommended outside some very specific usecases, some of which are mentioned above. After all, Wicket is component-oriented for a reason. I will not be going over the reasons why using components is better than writing out markup directly, those should be obvious to the reader. However, we take this approach here to show that Wicket is flexible and grants developers capability to do things in ways that may not necessarily conform to the best practices of the framework.

Create a class to model a menu item:

MenuItem.java

```
public class MenuItem implements Serializable {
    private final IModel<String> label;
    private final Class<? extends Page> page;

    public MenuItem(IModel<String> label,
        Class<? extends Page> page) {
        this.label = label;
        this.page = page;
    }

    // getters for label and page
}
```

Create an interface that pages can use to identify which menu item they belong to so the menu component knows which item is selected:

MenuPage.java

```
public interface MenuPage {
    MenuItem getItem();
}
```

Create the menu component:

Menu.java

```
public class Menu extends WebComponent {
    public Menu(String id,
```

```

        IModel<List<? extends MenuItem>> items) {
            super(id, items);
        }

    protected void onComponentTagBody(
        MarkupStream markupStream, ComponentTag openTag) {
        StringBuilder markup = new StringBuilder();
        renderMenu(markup);
        replaceComponentTagBody(markupStream, openTag, markup);
    }

    private void renderMenu(StringBuilder markup) {
        List<? extends MenuItem> items =
            (List<? extends MenuItem>) getDefaultModelObject();
        markup.append("<ul>");

        MenuItem current = null;
        if (getPage() instanceof MenuPage) {
            current = ((MenuPage) getPage()).getItem();
        }

        for (MenuItem item : items) {
            CharSequence url = urlFor(item.getPage(), null);

            String label=
                Strings.escapeMarkup(item.getLabel().getObject());

            item.getLabel().detach();

            markup.append("<li>");
            markup.append("<a ");

            markup.append("href='").append(url).append("'");

            if (item.equals(current)) {
                markup.append(" class='selected'");
            }

            markup.append(">");
            markup.append("<span>")
                .append(label)
                .append("</span>");

            markup.append("</a>");
            markup.append("</li>");
        }

        markup.append("</ul>");
    }
}

```

Define the menu items and add the menu in the BasePage:

BasePage.html

```
<html>
  <body>
    <div wicket:id="menu"></div>
    <wicket:child/>
  </body>
</html>
```

BasePage.java

```
public abstract class BasePage extends WebPage
    implements MenuPage {
```



Note that our BasePage contains menu items that reference subclasses of the BasePage. This may be seen as breaking the Open/Closed principle ([http://en.wikipedia.org/wiki/Open/closed\\_principle](http://en.wikipedia.org/wiki/Open/closed_principle)) and should not be done this way in a proper application; however, it is done this way here to keep things as short and as simple as possible.

```
    protected static MenuItem HOME =
        new MenuItem(Model.of("Home"),    HomePage.class);
    protected static MenuItem PRODUCTS = new MenuItem(Model.
of("Products"),
        ProductsPage.class);
    protected static MenuItem ABOUTUS = new MenuItem(Model.of("About
Us"),
        AboutUsPage.class);
    public BasePage() {
        add(new Menu("menu", new MenuModel()));
    }
    private static class MenuModel
        extends LoadableDetachableModel
        <List<? Extends MenuItem>> {
        protected List<? extends MenuItem> load() {
            List<MenuItem> items=new ArrayList<MenuItem>();
            items.add(HOME);
            items.add(PRODUCTS);
            items.add(ABOUTUS);
            return items;
        }
    }
}
```

Modify main entry pages to specify which menu item they belong to:

HomePage.java

```
public class HomePage extends BasePage {
    public MenuItem getItem() { return HOME; }
}
```

ProductsPage.java

```
public class ProductsPage extends BasePage {
    public MenuItem getItem() { return PRODUCTS; }
}
```

AboutUsPage.java

```
public class AboutUsPage extends BasePage {
    public MenuItem getItem() { return ABOUTUS; }
}
```

## How it works...

As our menu navigates between bookmarkable pages, our menu item needs to know the class of the page it will link to:

MenuItem.java

```
public class MenuItem implements Serializable {
    private final IModel<String> label;
    private final Class<? extends Page> page;
}
```

We create all menu items in the base page since this is the page that will contain the menu:

```
public abstract class BasePage extends WebPage
    implements MenuPage {
    protected static MenuItem HOME =
        new MenuItem(Model.of("Home"), HomePage.class);
    ...
}
```

Because the menu needs to know which menu item should be rendered in the selected state we create an interface that allows each page to specify that:

MenuPage.java

```
public interface MenuPage {
    MenuItem getItem();
}
```

As the `BasePage` represents the base for pages that will contain the menu it implements the interface forcing all of its subclasses to specify which menu item should be selected when that page is shown. Subclasses use menu items defined in the `BasePage` to identify the menu item:

`HomePage.java`

```
public class HomePage extends BasePage {  
    public MenuItem getItem() { return HOME; }  
}
```

The only missing piece now is the component that renders the menu. Since the menu is a simple unordered list we are going to render the markup ourselves rather than creating it using higher level components such as `ListViews` and `Labels`. We begin by creating a simple `WebComponent` that will replace the body of its tag with whatever markup we generate:

`Menu.java`

```
public class Menu extends WebComponent {  
    public Menu(String id,  
        IModel<List<? extends MenuItem>> items) {  
        super(id, items);  
    }  
  
    protected void onComponentTagBody(  
        MarkupStream markupStream, ComponentTag openTag) {  
        StringBuilder markup = new StringBuilder();  
        renderMenu(markup);  
        replaceComponentTagBody(markupStream, openTag, markup);  
    }  
}
```

All that's left is to implement the `renderMenu()` method which generates the unordered list markup. The full implementation of the method can be seen previously; here we will go over a couple of interesting bits.

To figure out the menu item that should be rendered in a selected state we query the current page:

`Menu.java`

```
MenuItem current = null;  
if (getPage() instanceof MenuPage) {  
    current = ((MenuPage) getPage()).getItem();  
}
```

Above we support the case when the page the menu is on does not implement the `MenuPage` interface; this helps the menu component to be more flexible.

Also note that when we output the label of the menu item into the markup we take care to escape it first:

```
String label=           Strings.escapeMarkup(item.getLabel()).
getObject();
```

As we are writing out raw markup ourselves we are responsible for what we output and have to perform any and all escaping ourselves. Escaping the menu item will help prevent XSS attacks in case the model returns some bit of string that can be manipulated by users of our application.

After we have retrieved the label from item's label model we take care to detach it:

```
item.getLabel().detach();
```

We detach the model ourselves because it is not bound to any component.

The rest of the `renderMenu()` method performs simple string manipulation to build the unordered list markup.

### More Info Section 1

Often web applications are organized hierarchically, for example the page navigation hierarchy can look as follows:



Lets modify our Menu component to support hierarchical navigation.

Modify MenuItem to support hierarchy:

MenuItem.java

```
public class MenuItem implements Serializable {
    private final IModel<String> label;
```

```
private final Class<? extends Page> page;
private MenuItem parent;
private MenuItem[] children;

public MenuItem(IModel<String> label,
    MenuItem... children) {
    this(label, null, children);
}

public MenuItem(IModel<String> label,
    Class<? extends Page> page) {
    this(label, page, new MenuItem[0]);
}

private MenuItem(IModel<String> label,
    Class<? extends Page> page, MenuItem... children) {
    this.label = label;
    this.page = page;
    this.children = children;
    for (MenuItem child : children) {
        child.parent = this;
    }
}
}
```

The menu item can now link to a page or to a list of child menu items. Let's modify the `BasePage` to create the new menu item hierarchy:

`BasePage.java`

```
public abstract class BasePage extends WebPage
    implements MenuPage {

    protected static MenuItem HOME = new MenuItem(Model.of("Home"),
        HomePage.class);

    protected static MenuItem PRODUCTS = new MenuItem(Model.
of("Products"),
        ProductsPage.class);

    protected static MenuItem STORE = new MenuItem(Model.of("About Our
Store"),
        StorePage.class);

    protected static MenuItem CONTACTUS = new MenuItem(Model.
of("Contact Us"),
        ContactUsPage.class);

    protected static MenuItem ABOUTUS = new MenuItem(Model.of("About
Us"),
        STORE, CONTACTUS);
}
```

```
protected static MenuItem ROOT = new MenuItem(null, HOME, PRODUCTS,
ABOUTUS);
```

Note that as `MenuItem` now supports hierarchy, we can define a `ROOT` item that contains the top-level entry levels and pass that to the `Menu` component rather than a list of top-level items.

Now we modify the `renderMenu()` method of our `Menu` component to render a hierarchical unordered list:

`Menu.java`

```
private void renderMenu(StringBuilder markup) {
    MenuItem item = (MenuItem) getDefaultModelObject();
    MenuItem current = null;
    if (getPage() instanceof MenuPage) {
        current = ((MenuPage) getPage()).getItem();
    }

    Set<MenuItem> selected = new HashSet<MenuItem>();
    while (current != null) {
        selected.add(current);
        current = current.getParent();
    }
    renderItem(markup, item, selected);
}

private void renderItem(StringBuilder markup, MenuItem item,
    Set<MenuItem> selected) {
    markup.append("<ul>");
    for (MenuItem child : item.getChildren()) {
        markup.append("<li>");

        // sample code renders the anchor tag for the menu item
        if (selected.contains(child)
            &&child.getChildren().length>0) {
            renderItem(markup, child, selected);
        }
        markup.append("</li>");
    }
    markup.append("</ul>");
}
}
```

The menu renders embedded unordered lists from the selected item to the top level items, so if we select an inner item such as **Contact Us** the rendered markup will look like this (URLs are omitted for brevity):

```
<ul>
  <li>
    <a><span>Home</span></a>
  </li>
  <li>
    <a><span>Products</span></a>
  </li>
  <li>
    <a class='selected'><span>About Us</span></a>
    <ul>
      <li>
        <a><span>About Our Store</span></a>
      </li>
      <li>
        <a class='selected'><span>ContactUs</span></a>
      </li>
    </ul>
  </li>
</ul>
```