Induction

High-performance, open-source, Java MVC Framework

www.inductionframework.org

Adinath Raveendra Raj

UJUG presentation
February 21, 2013

Once upon a time...

- I used to complain a lot about web frameworks…☺
- Most Java web frameworks were not addressing common cases with simple metaphors…
- In fact, in early 2000…I thought that one web framework I saw was just a prank…unfortunately for millions of developers…it was not
- In 2008, I needed an MVC framework and key issues remained unaddressed…it just got real…

What is Induction?

- A request-based Java MVC framework
- Painstakingly designed to be simple and powerful
- Designed to address the needs of large, complex web applications
- Requires a Servlet container

How is Induction different?

- Effectively build large web apps
  - Static MVC dependency analysis
  - Strongly typed redirects
  - Template data scoping
- Refactor fearlessly
  - Use your favorite IDE, it will find the MVC dependencies
  - Redirects and embedded URLs update automatically
- Develop faster
  - Just compile and refresh your page – dynamic class reloading
- Leverage powerful capabilities
  - Short URLs – without hardcoding
  - Multi-stage request interceptors
  - Response stream buffering
  - Replaceable templating engine, configuration loader

Static MVC dependency analysis

- The goal is to handle the following use-cases:
  - I want to refactor this controller, which pages may break?
  - I want to refactor this service (model), which controllers (or views) may break?
  - I want to refactor this page, which controllers may break?
- The solution:
  - A Controller is a Java class
  - A View is a Java class
  - A Model is a Java class (or interface)
  - All references between the above are strongly typed

Strongly typed redirects

- A controller initiates a redirect by *returning* a *redirect* object
- Here is what the code looks like:
  - *return new Redirect(LoginPage.class)*
- Induction passes the Redirect object to the Redirect Resolver which builds a URL

Template data scoping

- The goal is to handle this use-case:
  - I want to know exactly what data a template uses, and what models that data comes from
- The solution:
  - The class Java class for the template is also a Java bean that declares the data used in the template
  - The only data available in the template is the Java bean properties

Dynamic class reloading

- The goal is to just compile your code and refresh your web page to see the change

- The solution:
  - Integrated dynamic classloader
    - No restart of the servlet container is required to deploy most changes to models, views and controllers
    - A model, view or controller is reloaded if the respective class or one of its dependencies have changed
    - Completely unplug this classloader in production

Controllers

- A controller is a class that implements the *Controller* marker interface
- Each public method of the controller can be mapped to a URL
- The method requests a model simply by declaring it as a formal parameter
- The return value of a controller is significant

Controller Example 1

```
package demoapp.helloworld1_app;

import com.acciente.induction.controller.Controller;
import com.acciente.induction.controller.Response;
import java.io.IOException;

/**
 * A very simple controller that does the customary "Hello World"
 */
public class HelloWorldController implements Controller {

   public void handler(Response response) throws IOException {
      reponse.setContentType( "text/plain" );
      reponse.out().println( "Hello World, using a simple println()" );
   }
}
```

Controller Example 2

```
/**
 * An example of a multi-action controller
 */
public class UserProfileController implements Controller {

   /**
    * Action to open a user profile
    */
   public void open(UserProfileApp userProfileApp, Form form) throws IOException {
      // some sample code follows ...
      userProfileApp.open( form.getString( "userId" ) );
   }

   /**
    * Action to save a user profile
    */
   public void save(UserProfileApp userProfileApp, Form form) throws IOException {
      // some sample code follows ...
      userProfileApp.setFirstName( form.getString( "firstName" ) );
      userProfileApp.setLastName( form.getString( "lastName" ) );
      userProfileApp.save();
   }
}
```

Views

- A view class must implement one of the following interfaces:
  - Template
  - Image
  - ImageStreamer
  - Text
- A new instance of the view is created for each web request

Text View Example

```
package demoapp.helloworld2_app;

import com.acciente.induction.view.Text;

public class HelloWorldView implements Text {

   public String getText() {
      return "Hello World, using a Text view";
   }

   public String getMimeType() {
      return "text/plain";
   }
}
```

Template View Example

```
package demoapp.helloworld3_app;

import com.acciente.induction.view.Template;

/**
 * A HelloWorld view using a freemarker tinplate
 */
public class HelloWorldView implements Template {
   // bean attributes
   public String getFirstName() {
      return "John Doe";
   }

   // template methods
   public String getTemplateName() {
      return "HelloWorld.ftl";
   }

   public String getMimeType() {
      return "text/html";
   }
}
```

Controller/View Example

```
package demoapp.helloworld2_app;

import com.acciente.induction.controller.Controller;

/**
 * A simple controller that uses a Text view
 * to display hello world
 */
public class HelloWorldController implements Controller {

   public Class handler() {
      return HelloWorldView.class;
   }
}
```

Models

- **Zero** coupling to the framework (*any* class can be a model)
- Implements pure application logic
- A configurable factory may be provided for each model class
- Declarative control of the lifecycle of model instances:
  - Static
  - Application
  - Session
  - Request

Declaring a model class

```
<model-defs>
  ...
  <model-def>
    <class>demoapp.models_app.BarModel</class>
    <scope>session</scope>
    <factory-class>demoapp.models_app.BarModelFactory</factory-class>
  </model-def>
  ...
</model-defs>
```

Model Instantiation

- Induction support two modes of model instantiation, one favors simplicity and the other favors flexibility
  - Method 1: Singleton Public Constructor
    - The singleton public constructor mode of model instantiation is assumed if no factory class is defined for the model. In this mode Induction creates the model object by calling the singleton public constructor of the model class
  - Method 2: Model Factory
    - Specifying a factory class provides the most control over model instantiation
    - A model factory does not implement any special interface, it simply needs to have a method named createModel
    - The createModel(…) method is called with parameter injection. The types supported for parameter injection into the createModel method is identical to that described for the singleton public constructor above

Model Scope

- The options for model scope are:
  - *Static*: one instance of the model is created per instance of the servlet container
  - *Application*: one instance of the model is created per instance of the servlet
  - *Session*: one instance of the model is created per HTTP session
  - *Request*: one instance of the model is created per HTTP request. This implies that a model would be created for each HTTP request, so use this type only when the model needs to be created and torn down for each HTTP request

Template Output Buffering

- The buffered output is written to the response only if the template completes with no errors
  - Enables the error handler to send a redirect, since nothing was written to the response
  - Users no longer long stack trace

- This uses gobs of memory, right?
  - No! Induction maintains a pool of buffers, resulting in an essentially finite memory footprint!

Demos / Q & A

- Let’s write some code…
- Questions