# **Induction** MVC Framework
# A brief introduction to release 1.3.1b/1.4.0b
(compares to release 1.1.4b)

Adinath Raveendra Raj
Acciente, LLC
Dec 11, 2009

# What is Induction?

- Induction is a powerful, high-performance, Java MVC web application framework
- Induction supports dynamic application reloading, type-based dependency injection and dependency analysis between models, views and controllers
- Induction has an extensible architecture including support for view, controller and redirect resolvers, request interceptors, error handlers, plug-in configuration loaders and plug-in templating engines
- Induction is compatible from JDK 1.4 to JDK 1.6 (Java 6)
- Induction is a new and compelling alternative to other web application frameworks including Struts 2 and Spring MVC
- Induction is open-source software released under the commercial friendly Apache License 2.0

# Why Induction?

- It should be possible to simply recompile the changed source (using your favorite IDE) and simply reload the respective web page.
- It should be possible to analyze the dependencies between models, views and controllers (this helps maintain complex web applications)
- It should be easy to leverage the power of dependency injection to wire the models, views and controllers
- It should not be cluttered by superfluous XML configuration files.
- It should be possible to decouple your web application from specific URL mappings
- Handling file uploads should be simple to the point of being unremarkable

# Dynamic Reloading

- Induction does not require restarting the servlet container 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
- The dependencies of a class are determined by examining the class bytecode
- Induction uses the reloading class loader in the Acciente Commons library
- The reloading class loader is written with performance optimization considerations

# MVC Dependency Analysis

- The ability to analyze dependencies between models, views and controllers reduces the cost of evolving complex applications
- Models, views and controllers are represented as Java classes to promote this analysis using an IDE
- This also promotes the use of an IDE's refactoring tools to when a model, view or controller is changed

# Template Data Dependency

- The goal of this design feature is to answer the questions:
  - What data does this template use?
  - Where model does this data come from?
- This is achieved by mapping the data space of of a template to a single Java bean
- Each template is required to have a single Java bean that documents the data used in that specific template

# Dependency Injection

- Induction uses type-based dependency injection
- Dependency injection is automatic in the following contexts:
  - Constructor formal parameter list for:
    - Models
    - Views
    - Controllers
    - Interceptors
  - Method formal parameter list for:
    - Controllers
    - Interceptors
- Requires only a declaration of the model class to Induction
- Includes environment values, e.g. javax.servlet.HttpServletResponse

# Views

- A view class is a that implements one of the following interfaces (based on the purpose of view):
  - Template
  - Image
  - ImageStreamer
  - Text
- A class, say C1, that implements the Template interface is processed via one of the integrated template engines.
  - When processing the template the only data passed to the template is an instance of the class C1. **C1 serves as the bean that documents precisely what data the associated template needs**.

# Template Views

- Template views are the most commonly used view type in most applications
- To be considered a view a class, say LoginPage, must implements Induction's Template interface
- When a template view is requested
  - An instance of the LoginPage class is created
  - The instance is handed to the template engine plug-in.
  - When processing the template the only data passed to the template is the LoginPage instance. **Our LoginPage class automatically documents precisely what data the associated template depends on**.

# 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";
   }
}
```

# 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";
   }
}
```

# 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()
   {
      // typically we would do some processing of the user input here
      // and pass some data into the view via its constructor

      return HelloWorldView.class;
   }
}
```

# What's new with views?

- Major view related features came in 1.2.0b
  - Direct view activation without a controller
    - This means there is a new view resolver interface
  - Support for managed view instantiation
    - dependency injection into the view constructor
  - Controllers may now return a view type
    - You can do "return MyView.class"
    - Previously: "return new MyView( ... ) )"

# Another cool feature in 1.4.0b

- Template Output Buffering
  - What's the big deal?
  - The buffered output is written to the response only if the template completes with no errors
  - So users will be spared the 3 page template error stack trace
  - Allows a error handler to send a redirect, since nothing was written to the response
  - This uses gobs of memory, right? That's the cool part!
  - Induction implements an essentially finite memory footprint buffering scheme
  - The buffering is done using buffer pools where the buffers get reused

# Also slated for 1.4.0b

- Automatic view-to-view injection support
  - The motivation is to support building composite views
  - Declaring a one view as a constructor parameter in another view will be automatically recognized by the dependency

```
public CatalogPage( ProductSelectorDiv productSelectorDiv )
{
    // catalog page code
   this.productSelectorDiv = productSelectorDiv;
}

public ProductSelectorDiv getProductSelectorDiv()
{
   return this.productSelectorDiv;
}
```

- More built-in support for composite views (still thinking about the design of this one)

# Models

- Implements pure application logic, completely independent of views and controllers (and the framework)
- **Any** class can be a model
- A configurable factory may be provided for each model class
- Declarative control of the lifecycle of each model class:
  - Application: a model object is created for the life of the application
  - Session: a model object is created for each browser session
  - Request: a model object is created for each controller invocation

# 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>
        <init-on-startup>false</init-on-startup>
    </model-def>
    ....
</model-defs>
```

# Model Instantiation

- Induction support two modes of model instantiation, one mode favors simplicity and the other provides complete control
  - 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 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:
  - *Application*: one instance of the model is created per instance of the Induction dispatcher 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.

# Model-to-Model Injection

- Model-to-model injection comes into play when you need an instance of one model, say A, in another model, say B
- Instead of explicitly instantiating an instance of model A in model B and managing its lifecycle, with no added configuration, Induction's parameter injection mechanism can be used to link the model objects
- The model objects that are linked could have different scopes. For example, model A may be application scope and model B may be session scope
- Induction injects an instance of one model class, say A, into the constructor of another model class, say B, based on the constructor's parameter list of model class B

# Model-to-Model Inj. example

```
package demoapp.models_app;

public class FooModel
{
   private BarModel _oBarModel;

   public FooModel( BarModel oBarModel )
   {
      _oBarModel = oBarModel;

      System.out.println( "FooModel: constructor called @time: " + System.currentTimeMillis() );
   }

   public BarModel getBarModel()
   {
      return _oBarModel;
   }
}
```

# Controllers

- A controller is a class which implements the Controller marker interface (the interface does not enforce any methods)
- A model object or environment object (such as the Request, Response or Form) is accessible in a controller by simply declaring a parameter of the model object's type in the controller's respective method (this is basically dependency injection)
- A controller may have a method for each distinct task that it performs
- 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 oReponse ) throws IOException
   {
      oReponse.setContentType( "text/plain" );
      oReponse.out().println( "Hello World, using a simple println()" );
   }
}
```

# Controller example 2a

```
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 HelloWorldView handler()
   {
      // typically we would do some processing of the user input here
      // and pass some data into the view via its constructor

      return new HelloWorldView();
   }
}
```

# Controller example 2b

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

# Controller example 3

```
/**
 * 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( );
   }
}
```

# Resolving URLs

- View Resolvers
  - A view resolver is a user provided handler that resolves a URL request to a view class name
  - View resolver were added in 1.2.0b since this version introduced direct view activation
- Controller Resolvers
  - A controller resolver is a user provided handler that resolves a URL request to a controller class name and handler method name
- Redirect Resolvers
  - A redirect resolver is a user provided handler that resolves a redirect request issued by a controller to an complete URL

# What's else is new with resolvers?

- Induction now ships with a built-in set of production quality resolvers, known as the ShortURL resolvers

- The ShortURL resolvers are the most powerful resolvers of any MVC framework we know of
  - You do not need to enumerate URL mappings for each of your views/controller
  - You do not need to splatter URLs all over your code!
  - You configuration is concise
  - The resolvers are fast

# The ShortURL resolvers

- It starts by auto-discovering your view and controllers
- You specify a regex that describes your class pattern
- The same regex should have a matching group that extracts a unique key from the class name
- You specify a package root to scan
- The auto-discovery mechanism is very fast (<1s for a large app) and memory efficient
  - It can check if class is view/controller without loading the class

# The ShortURL resolvers

- Next you specify the URL pattern that you are going to use
  - You specify a regex that describes your URL pattern
  - The same regex should have a matching group that extracts a unique key from the URL
  - This key extracted from the URL is used as an index into the keys extracted from the classnames
- Short URL mappings with a lot of flexibility and very little manual work...done!

# The ShortURL resolvers

Let's look at how controllers are mapped:

```
<controller-mapping>

    <url-to-class-map>
        <url-pattern>/(\w+)(?:\.(\w+))?\.action</url-pattern>
        <class-packages>demoapp</class-packages>
        <class-pattern>(?:.*\.)?(\w*)Controller</class-pattern>
    </url-to-class-map>

    <default-handler-method>handler</default-handler-method>
    <ignore-handler-method-case>true</ignore-handler-method-case>

</controller-mapping>
```

# Request Interceptors

- What is a Request Interceptor?
  - A request interceptor is a piece of code that gets activated for every single HTTP request received by your application
  - Interceptors are very useful when you need to perform some common processing for every HTTP request
- Request Interceptors are new in 1.3.0b

# Request Interceptors

- What does a Request Interceptor look like?
  - A request interceptor is simply an ordinary Java class that implements the RequestInterceptor interface
  - Induction looks for the methods named:
  - preResolution( … )
  - postResolution( … )
  - preResponse( … )
  - postResponse( … )
  - If it finds one or more of these methods they get called at different points in the request processing cycle

# Request Interceptors

- preResolution(...)
  - this interceptor method is invoked PRIOR to attempting to resolve the target of the HTTP request using the controller and view resolvers.
- postResolution(...)
  - this interceptor method is invoked AFTER attempting to resolve the target of the HTTP request using the controller and view resolvers.
- preResponse(...)
  - this interceptor method called as follows:
    - if the request resolved to a controller then this interceptor method is called AFTER the controller is executed but BEFORE the view or redirect object returned by the controller is processed.
    - if the request resolved to a view then this interceptor method is called BEFORE the view is processed.
- postResponse(...)
  - this interceptor method is invoked AFTER processing a view or redirect object

# What's in the cards?

- Pluggable IoC provider
  - Guice
  - Spring IoC
- Better support for composite views
- Plug-ins for other view technologies
- More built-in AJAX support
  - Built-in support for JSON views
- Test compatibility Google App Engine
  - Currently GAE does not support custom classloader based on the SecureClassLoader

# Demo / Q & A

- Time for some demos:
  - HelloWorld
  - Counter
  - Redirect
  - FileUpload
  - Models

- ....and time to to ask questions like.....where can I download Induction? ☺