Spring MVC View Resolvers

"Code with Passion!"

Topics

- Resolving views
- Spring-provided view resolvers
- Chaining view resolvers
- Views vs @ResponseBody
- Automatic logical view name generation
- ViewController & RedirectViewController
- ContentNegotiatingViewResolver

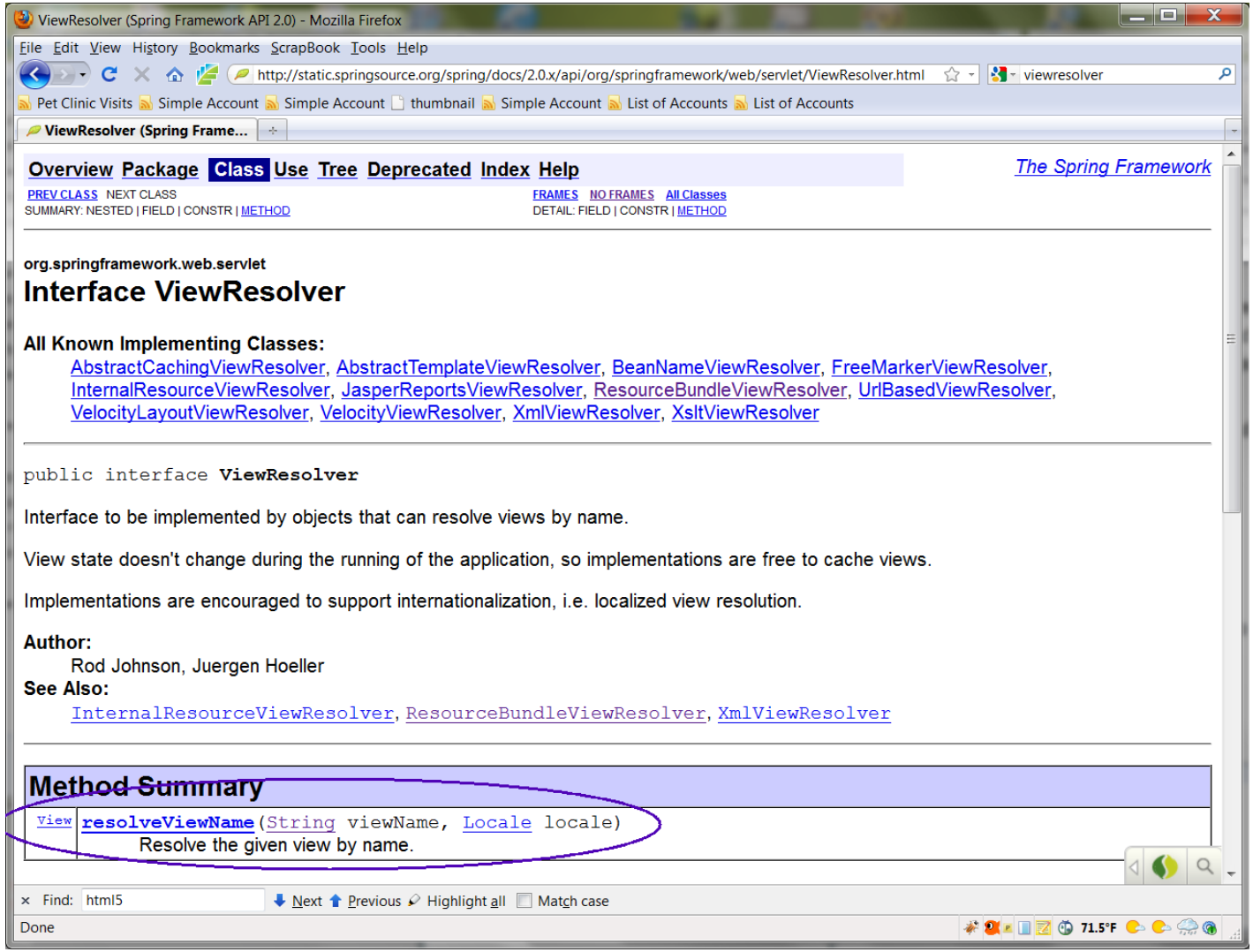
Resolving Views

Resolving Views

- All handler methods in a controller must resolve to a logical view name
  - Explicitly (by returning a String, View, or ModelAndView) or
  - Implicitly (void return type)
- A logical view name is then resolved by a view resolver
  - "Resolving a view" means creating an appropriate View object
  - Spring MVC goes through a series of view resolvers (in sequence) until it find a view resolver which can handle the logical view name
  - Each view resolver returns a View object if it can handle it or null otherwise - if null returns, next view resolver is consulted
- Spring comes with a set of view resolvers
  - Every view resolver implements *ViewResolver* interface

    View resolveViewName(String viewName, Locale locale)

Interface ViewResolver



View Interface

- All implementations of this interface are responsible for
  - Rendering content
  - Exposing the model - A single view exposes multiple model attributes
- Spring provides many implementations of View interface
  - JstlView (for displaying JSP)
  - MappingJackson2JsonView (for displaying JSON)
  - MappingJackson2XmlView (for displaying XML)
  - TilesView (for displaying Tiles)
  - ThymeleafView (for displaying Thymeleaf)
  - FreeMarkerView (for displaying FreeMaker)
  - VelocityView (for displaying Velocity)

Spring-Provided ViewResolvers (focused on JSP)

View Resolvers

- UrlBasedViewResolver & InternalResourceViewResolver
  - InternalResourceViewResolver is a subclass of UrlBasedViewResolver
- BeanNameViewResolver
- ResourceBundleViewResolver
- ContentNegotiatingViewResolver
  - This is a special view resolver in the sense that it selects a view resolver and delegates the view resolution to the selected view resolver

UrlBasedViewResolver

// A simple implementation of the ViewResolver interface that effects the direct
// resolution of logical view names to URLs.

// Example: Suppose test is returned as a logical view name, /WEB-INF/jsp/test.jsp
// gets selected as a view

```
// Configuration of the UrlBasedViewResolver
@Bean
public UrlBasedViewResolver getViewResolver() {
    UrlBasedViewResolver viewResolver = new UrlBasedViewResolver();
    viewResolver.setViewClass(JstlView.class)  // The View type is JstlView
    viewResolver.setPrefix("/WEB-INF/views/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
}
```

InternalResourceViewResolver

```
// A convenience subclass of UrlBasedViewResolver that supports
// InternalResourceView (i.e. Servlets and JSPs), and subclasses such
// as JstlView and TilesView.

@Bean
public InternalResourceViewResolver getViewResolver() {
    InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
    viewResolver.setPrefix("/WEB-INF/views/");
    viewResolver.setSuffix(".jsp");
    return viewResolver;
}
```

BeanNameViewResolver

```
// Simple implementation of ViewResolver that interprets a view name as bean
// name in the current application context.


@Bean
public BeanNameViewResolver beanViewResolver() {
   BeanNameViewResolver resolver = new BeanNameViewResolver();
   resolver.setOrder(1);
   return resolver;
}

// Suppose a handler returns “welcome” as the logical view name, the
// BeanNameViewResolver will retrieve the View bean from the context and use
// this to render the JSP

@Bean(name="welcome")
public JstlView getJstlView(){
   JstlView jstlView = new JstlView();
   jstlView.setUrl("/WEB-INF/views/welcome.jsp");
   return jstlView;
}
```

Chaining ViewResolvers

Chaining ViewResolver's

- You can chain view resolvers by configuring more than one resolver to your application context and, if necessary, by setting the *order* property to specify ordering.
  - The higher the order property, the later the view resolver is positioned in the chain
  - *InternalResourceViewResolver* is automatically positioned as the last ViewResolver (if it does not have its own *order* property set)
- If a specific view resolver does not return a View object (in other words, returns null), Spring continues to call remaining view resolvers until a view is resolved

Chaining ViewResolver's

```
// In the following example, the chain of view resolvers consists of two resolvers,
// an InternalResourceViewResolver, which is always automatically positioned as
// the last resolver in the chain anyway, and an BeanNameViewResolver

@Bean
public BeanNameViewResolver beanViewResolver() {
    BeanNameViewResolver resolver = new BeanNameViewResolver();
    resolver.setOrder(1);
    return resolver;
}

@Bean
public InternalResourceViewResolver internalResourceViewResolver() {
    InternalResourceViewResolver resolver = new InternalResourceViewResolver();
    resolver.setPrefix("/WEB-INF/myviewdir/");
    resolver.setSuffix(".jsp");
    return resolver;
}
```

Lab:

Exercise 1: BeanNameViewResolver

4950_spring4_mvc_view_resolvers.zip

Views vs. @ResponseBody

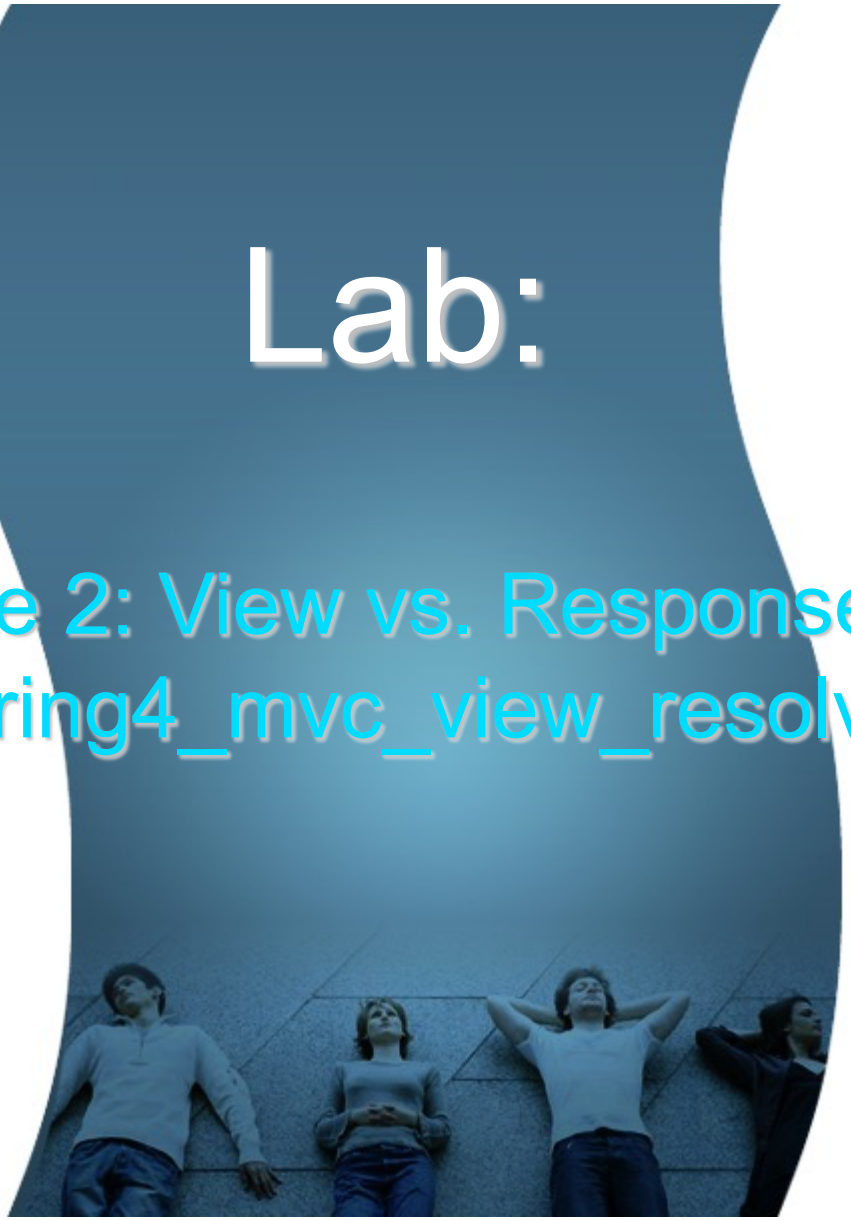
Two Schemes of Rendering Response

- Two schemes for rendering response (in the controller method)
  - Scheme #1: ViewResolver + View
  - Scheme #2: HttpMessageConverter
- They are triggered in different ways
  - Scheme #1: Render a view by returning a logical view String
  - Scheme #2: Write a message directly by returning a *@ResponseBody* or *ResponseEntity*
- Which one to use?
  - Use ViewResolver + View to generate documents for display in a web browser - HTML, PDF, etc
  - Use @ResponseBody or ResponseEntity to exchange data with web service clients - JSON, XML, etc

Lab:

Exercise 2: View vs. ResponseBody

4950_spring4_mvc_view_resolvers.zip



Automatic Logical View Name Generation

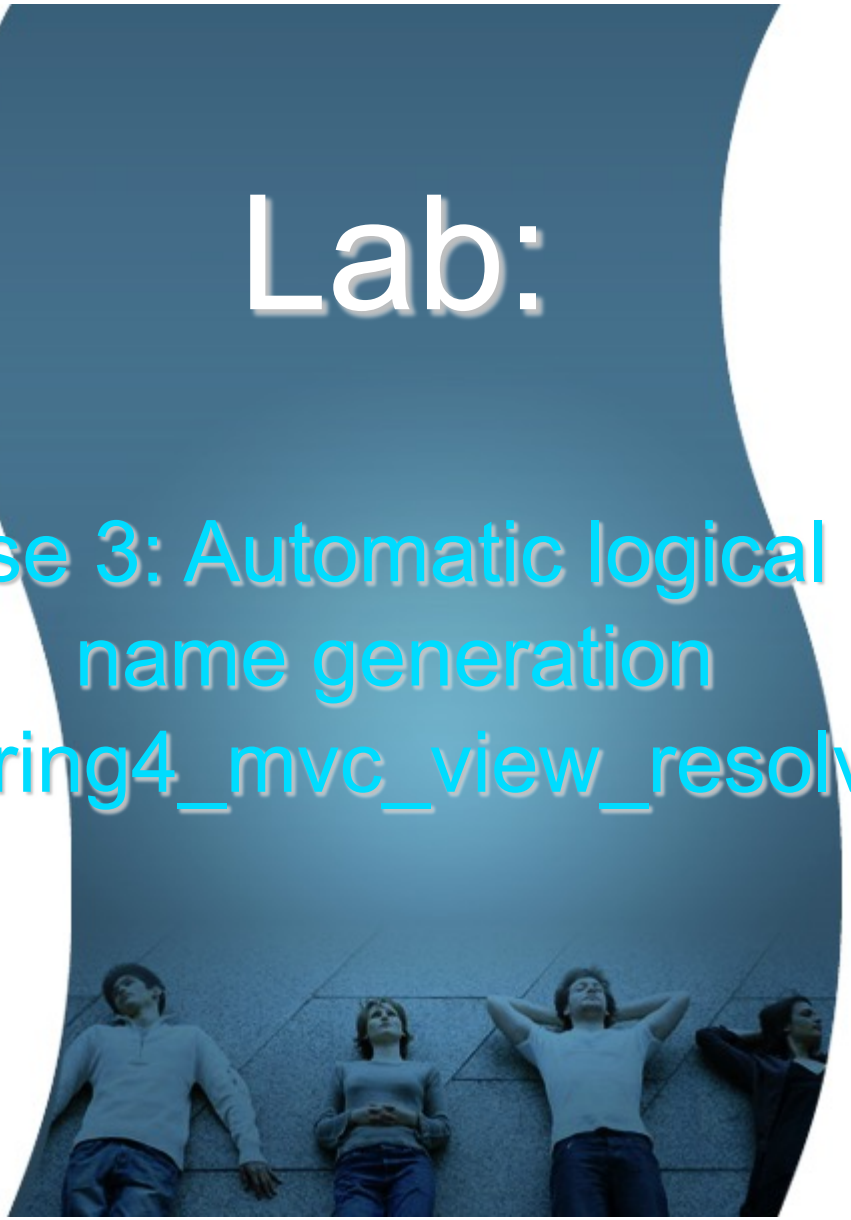
Handler Without Setting Logical View

```
// A request URL of http://localhost/registration.html results in a logical view
// name of registration

public class RegistrationController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
                                      HttpServletResponse response) {
        // No view is set
        ModelAndView mav = new ModelAndView();
        // add data as necessary to the model...
        return mav;
        // notice that no View or logical view name has been set
    }
}
```

Lab:

Exercise 3: Automatic logical view name generation

4950_spring4_mvc_view_resolvers.zip



ViewController & RedirectViewController Configuration

ViewController Configuration

- Used to immediately map incoming request to a view - no controller is involved

```
WebMvcConfigurerAdapter mvcViewConfigurer() {
    return new WebMvcConfigurerAdapter() {

        @Override
        public void addViewControllers(ViewControllerRegistry registry) {
            registry.addViewController("/").setViewName("home");
            registry.addViewController("/myhome").setViewName("myhome");
            registry.addViewController("/mykitchen.html").setViewName("kitchen");
        }
    };
}
```

RedirectViewController Configuration

- Used to redirect an incoming request immediately

```
WebMvcConfigurerAdapter mvcViewConfigurer() {
    return new WebMvcConfigurerAdapter() {

        @Override
        public void addViewControllers(ViewControllerRegistry registry) {
            registry.addRedirectViewController("/", "/welcome");
            registry.addRedirectViewController("/myurl", "/my-redirected-url");
        }
    };
}
```

Lab:

Exercise 4: ViewController & RedirectViewController

4950_spring4_mvc_view_resolvers.zip

ContentNegotiating ViewResolver

What is Content Negotiation?

- Client can ask the desired representation (of the response) through
  - URL extension (more common)
  - *Accept* HTTP header
- Example URL extensions
  - http://example.com/hotels.xml
  - http://example.com/hotels.json
  - http://example.com/hotels.html
  - http://example.com/hotels.pdf
- *ContentNegotiatingViewResolver*
  - It wraps one or more other ViewResolvers, looks at the Accept header or file extension, and resolves a view accordingly

What is ContentNegotiatingViewResolver?

- The *ContentNegotiatingViewResolver* does not resolve views itself
  - It delegates to other view resolvers
- A selected view resolver resolves a logical view name (returned from a controller method) to a View object
  - Different view resolvers resolve a logical view name into different View objects

Two Schemes for the Client for asking particular View type

- Scheme #1 - Use a file extension
  - The URI *http://www.example.com/users/fred.pdf* requests a PDF representation of the user *fred*
  - The URI *http://www.example.com/users/fred.xml* requests an XML representation of the user *fred*
- Scheme #2 - Set the Accept HTTP request header to list the media types (less common than scheme #1 since it is harder to set Accept HTTP header)
  - HTTP request for *http://www.example.com/users/fred* with an Accept header set to *application/pdf* requests a PDF representation of the user *fred*
  - HTTP request for *http://www.example.com/users/fred* with an Accept header set to *text/xml* requests an XML representation

ContentNegotiatingViewResolver Configuration

```
@Bean
public ViewResolver contentNegotiatingViewResolver(ContentNegotiationManager manager) {

    ContentNegotiatingViewResolver resolver = new ContentNegotiatingViewResolver();
    resolver.setContentNegotiationManager(manager);

    // Define all possible view resolvers
    List<ViewResolver> resolvers = new ArrayList<ViewResolver>();

    resolvers.add(jspViewResolver());
    resolvers.add(jsonViewResolver());
    resolvers.add(jaxb2MarshallingXmlViewResolver());
    resolvers.add(pdfViewResolver());
    resolvers.add(excelViewResolver());

    resolver.setViewResolvers(resolvers);
    return resolver;
}
```

Lab:

Exercise 4: ContentNegotiatingViewResolver

4950_spring4_mvc_view_resolvers.zip

Code with Passion!