# Spring MVC REST

Sang Shin
JPassion.com
"Learn with Passion!"

# Topics

- Annotation-based REST support
- URI template
- Content negotiation
- Feed views
- XML marshaling views
- JSON mapping views
- HTTP DELETE/PUT method support
- RestTemplate
- Exception handling
- Object to/from HTTP Message conversion

# Annotation-based REST Support

# Spring REST Support

- Closely tied with Annotation-model Spring MVC
- Issues to be dealt with with REST handling
  - URI templates
  - Content negotiation
  - HTTP DELETE/PUT method support (HTTP method conversion)

# URI Template

# URI Template

- Works the same as in Spring MVC
  - A URI template is a URI-like string, containing one or more variable names. When these variables are substituted for values, the template becomes a URI
  - Use of URI templates is supported through the *@PathVariable* annotation

```
@RequestMapping("/hotels/{hotelId}")
public String getHotel(@PathVariable String hotelId, Model model) {
    Hotel hotel = hotelService.getHotel(hotelId);
    model.addAttribute("hotel", hotel);
    return "hotelDetail";
}
```

# URI Template Example Code

- Can handle chained path, for example, /hotels/2/bookings/4 or /hotels/1/bookings/3

```
@RequestMapping(value="/hotels/{hotel}/bookings/{booking}",
method=RequestMethod.GET)
public String getBooking(@PathVariable("hotel") long hotelId,
@PathVariable("booking") long bookingId, Model model) {
    Hotel hotel = hotelService.getHotel(hotelId);
    Booking booking = hotel.getBooking(bookingId);
    model.addAttribute("booking", booking);
    return "booking";
}
```

# URI Template Example Code

- Can handle Ant style path – wildcard *

```
@RequestMapping(value="/hotels/*/bookings/{booking}",
                        method=RequestMethod.GET)
public String getBooking(@PathVariable("booking") long bookingId, Model model) {
    ...
}
```

# URI Template Example Code

- Can use data binding

```
@InitBinder
public void initBinder(WebDataBinder binder) {
    SimpleDateFormat dateFormat =
            new SimpleDateFormat("yyyy-MM-dd");
    binder.registerCustomEditor(Date.class,
            new CustomDateEditor(dateFormat, false));
}

@RequestMapping("/hotels/{hotel}/dates/{date}")
public void date(@PathVariable("hotel") String hotel,
                 @PathVariable Date date) {
    ...
}
```

# Lab:

## Exercise 1: Simple REST Request Handling
## 4949_spring3_mvc_rest.zip

# Content Negotiation

# What is Content Negotiation?

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

# 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 to a View object

# 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 that it understands (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

# “mediaTypes” Property of ContentNegotiatingViewResolver

- To support the resolution of a view based on a file extension (scheme #1), use the *ContentNegotiatingViewResolver* bean property *mediaTypes* to specify a mapping of file extensions to media types

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
  <property name="mediaTypes">
    <map>
      <entry key="atom" value="application/atom+xml"/>
      <entry key="html" value="text/html"/>
      <entry key="json" value="application/json"/>
      <entry key="pdf" value="application/pdf"/>
    </map>
  </property>
  ....
</bean>
```

# Example: ContentNegotiatingViewResolver

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
 ...
 <property name="viewResolvers">
  <list>
   <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
   <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/"/>
    <property name="suffix" value=".jsp"/>
   </bean>
  </list>
 </property>
```

- The *ContentNegotiatingViewResolver* selects an appropriate View Resolver to handle the request by comparing the request media type(s) with the media type (also known as Content-Type) supported by the View associated with each of its ViewResolvers.
- If *ContentNegotiatingViewResolver's* list of ViewResolvers is not configured explicitly, it automatically uses any ViewResolvers defined in the application context.

# Default View Resolver

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
 ...
 <property name="defaultViews">
  <list>
    <bean class="org.springframework.web.servlet.view.json.MappingJacksonJsonView" />
  </list>
 </property>
```

- If a compatible view cannot be supplied by the ViewResolver chain, then the list of views specified through the *DefaultViews* property will be consulted

# Controller Code Still Returns Logical View

```
// The corresponding controller code that returns an Atom RSS feed for a URI of the form
//  http://localhost/content.atom or http://localhost/content with an Accept header of
//  application/atom+xml.  Note that it still returns a logical view. There is no view
//  specific code in the controller.
@Controller
public class ContentController {

    private List<SampleContent> contentList = new ArrayList<SampleContent>();

    @RequestMapping(value="/content", method=RequestMethod.GET)
    public ModelAndView getContent() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("content");
        mav.addObject("sampleContentList", contentList);
        return mav;
    }

}
```

# Complete Configuration File

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
  <property name="mediaTypes">
    <map>
      <entry key="atom" value="application/atom+xml"/>
      <entry key="html" value="text/html"/>
      <entry key="json" value="application/json"/>
    </map>
  </property>
  <property name="viewResolvers">
    <list>
      <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
      <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
      </bean>
    </list>
  </property>
  <property name="defaultViews">
    <list>
      <bean class="org.springframework.web.servlet.view.json.MappingJacksonJsonView" />
    </list>
  </property>
</bean>
```

```
<bean id="content" class="com.springsource.samples.rest.SampleContentAtomView"/>
```

# Explanation of Previous Slide's Example

- If a request is made with an *.html*l extension, the view resolver looks for a view that matches the text/html media type.
  - The *InternalResourceViewResolver* provides the matching view for text/html.
- If the request is made with the file extension *.atom*, the view resolver looks for a view that matches the *application/atom+xml* media type.
  - This view is provided by the *BeanNameViewResolver* that maps to the *SampleContentAtomView* if the view name returned is "*content*".
- If the request is made with the file extension *.json*, the *MappingJacksonJsonView* instance from the *DefaultViews* list will be selected regardless of the view name.

# JSON Mapping View

# JSON Mapping View

- The *MappingJacksonJsonView* uses the Jackson library's *ObjectMapper* to render the response content as JSON.
- By default, the entire contents of the model map (with the exception of framework-specific classes) will be encoded as JSON.
- For cases where the contents of the map need to be filtered, users may specify a specific set of model attributes to encode via the *RenderedAttributes* property.

# Configuration

```
<!-- Views mapped in views.properties (PDF, XLS classes, and others) -->
<bean id="contentNegotiatingResolver"
   class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
   <property name="mediaTypes">
     <map>
       <entry key="html" value="text/html" />
       <entry key="xml" value="application/xml" />
       <entry key="json" value="application/json" />
     </map>
   </property>
</bean>

<bean id="beanNameViewResolver"
       class="org.springframework.web.servlet.view.BeanNameViewResolver">
</bean>

<!-- Logical view "accounts" gets handled by "MappingJacksonJsonView" class -->
<bean id="accounts"
        class="org.springframework.web.servlet.view.json.MappingJacksonJsonView" />
```

# Lab:

## Exercise 2: Displaying response data in JSON

## 4949_spring3_mvc_rest.zip

# XML Marshalling View

# XML Marshalling View

- The MarhsallingView uses an XML Marshaller defined in the *org.springframework.oxm* package to render the response content as XML.
- The object to be marshalled can be set explicitly using *MarhsallingView's* modelKey bean property.

# JAXB2 Marshaller

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
  <property name="mediaTypes">
     <map>
        <entry key="xml" value="application/xml"/>
        <entry key="html" value="text/html"/>
     </map>
  </property>
  <property name="viewResolvers">
     <list>
        <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
        <bean class="org.springframework.web.servlet.view.UrlBasedViewResolver">
            <property name="viewClass" value="org.springframework.web.servlet.view.tiles2.TilesView"/>
        </bean>
     </list>
  </property>
</bean>

<bean id="studentsView" class="org.springframework.web.servlet.view.xml.MarshallingView">
   <constructor-arg ref="jaxbMarshaller" />
</bean>

<!-- JAXB2 marshaller. Automagically turns beans into xml -->
<bean id="jaxbMarshaller" class="org.springframework.oxm.jaxb.Jaxb2Marshaller">
   <property name="classesToBeBound">
         <list>
               <value>com.spring.datasource.Classroom</value>
               <value>com.spring.datasource.Student</value>
         </list>
   </property>
</bean>
```

# XStream Marshaller

```
<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">
  <property name="mediaTypes">
     <map>
        <entry key="xml" value="application/xml"/>
        <entry key="html" value="text/html"/>
     </map>
  </property>
  <property name="viewResolvers">
     <list>
        <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
        <bean class="org.springframework.web.servlet.view.UrlBasedViewResolver">
            <property name="viewClass" value="org.springframework.web.servlet.view.tiles2.TilesView"/>
        </bean>
     </list>
  </property>
</bean>
<!-- Use bean name as the view resolver -->
<bean class="org.springframework.web.servlet.view.BeanNameViewResolver" />

<!-- "bookXmlView" logical view name is displayed as XML -->
<bean id="bookXmlView"
    class="org.springframework.web.servlet.view.xml.MarshallingView">
    <constructor-arg>
          <bean class="org.springframework.oxm.xstream.XStreamMarshaller">
          </bean>
    </constructor-arg>
</bean>
```

# Lab:

## Exercis3: Display Response in XML
## 4949_spring3_mvc_rest.zip

# Feed Views (Atom, RSS)

# Feed Views

- Both *AbstractAtomFeedView* and *AbstractRssFeedView* inherit from the base class *AbstractFeedView* and are used to provide Atom and RSS Feed views respectfully.
- They are based on java.net's ROME project

# Creating Atom

```
// AbstractAtomFeedView requires you to implement the buildFeedEntries()
// method and optionally override the buildFeedMetadata() method
// (the default implementation is empty)
public class SampleContentAtomView extends AbstractAtomFeedView {

  @Override
  protected void buildFeedMetadata(Map<String, Object> model, Feed feed,
      HttpServletRequest request) {
    // implementation omitted
  }

  @Override
  protected List<Entry> buildFeedEntries(Map<String, Object> model,
      HttpServletRequest request, HttpServletResponse response)
      throws Exception {

    // implementation omitted
  }
}
```

# Creating Rss

```
public class SampleContentRssView extends AbstractRssFeedView {

  @Override
  protected void buildFeedMetadata(Map<String, Object> model, Channel feed,
                                   HttpServletRequest request) {
    // implementation omitted
  }

  @Override
  protected List<Item> buildFeedItems(Map<String, Object> model,
      HttpServletRequest request, HttpServletResponse response)
      throws Exception {
    // implementation omitted
  }

}
```

# Lab:

## Exercise 4: Displaying response data in Atom/Rss

## 4949_spring3_mvc_rest.zip

# RestTemplate

# Why RestTemplate?

- Invoking RESTful services in Java is typically done using a helper class such as Jakarta Commons *HttpClient*. For common REST operations this approach is too low level as shown below

```
String uri = "http://example.com/hotels/1/bookings";

PostMethod post = new PostMethod(uri);
String request = // create booking request content
post.setRequestEntity(new StringRequestEntity(request));

httpClient.executeMethod(post);

if (HttpStatus.SC_CREATED == post.getStatusCode()) {
  Header location = post.getRequestHeader("Location");
  if (location != null) {
    System.out.println("Created new booking at :" + location.getValue());
  }
}
```

# RestTemplate

- *RestTemplate* provides higher level methods that correspond to each of the six main HTTP methods that make invoking many RESTful services a one-liner and enforce REST best practices
- RestTemplate methods
  - getForObject(), getForEntity() for HTTP GET
  - postForLocation (String url, ...) and postForObject (String url, ...) for HTTP POST
  - delete() for HTTP DELETE
  - put(String url, ...) for HTTP PUT
  - headForHeaders(String url, ..) for HTTP HEAD
  - optionsForAllow(String url, ..) for HTTP OPTIONS

# Method Naming Convention

- *getForObject()* will perform a GET, convert the HTTP response into an object type of your choice and return that object
- *postForLocation()* will do a POST, converting the given object into a HTTP request and return the response HTTP Location header where the newly created object can be found

# URI Template Arguments

- Each method takes URI template arguments in two forms
  - String variable length argument

```
String result = restTemplate.getForObject(
        "http://example.com/hotels/{hotel}/bookings/{booking}",
        String.class,"42", "21");
```

  - Map<String, String>

```
Map<String, String> variables = new HashMap<String, String>(2);
variables.put("hotel", "42");
variables.put("booking", "21");
String result = restTemplate.getForObject(
        "http://example.com/hotels/{hotel}/rooms/{booking}",
         String.class,
         variables);
```

# Lab:

## Exercise 5: RestTemplate
## 4949_spring3_mvc_rest.zip

# HTTP PUT/DELETE Method Support

# HTTP PUT/DELETE Method support

- Another key principle of REST is the use of the Uniform Interface.
  - All resources (URLs) can be manipulated using the same four HTTP method: GET, PUT, POST, and DELETE
- While HTTP defines these four methods, HTML only supports two: GET and POST
- Two schemes to support PUT and DELETE
  - Scheme #1: Use JavaScript to do your PUT or DELETE
  - Scheme #2: Do a POST with the 'real' method as an additional parameter (modeled as a hidden input field in an HTML form).

# Scheme #2 - Do POST with real

- web.xml configuration

```
<!-- Filter that converts posted method parameters into HTTP methods, retrievable
       via HttpServletRequest.getMethod(). Since browsers currently only support GET
       and POST, a common technique - used by the Prototype library, for instance -
       is to use a normal POST with an additional hidden form field (_method) to
       pass the "real" HTTP method along. This filter reads that parameter and
       changes the HttpServletRequestWrapper.getMethod() return value accordingly.  -->
<filter>
       <filter-name>httpMethodFilter</filter-name>
       <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
</filter>

<filter-mapping>
       <filter-name>httpMethodFilter</filter-name>
       <servlet-name>petclinic</servlet-name>
</filter-mapping>
```

# Scheme #2: Method conversion in the Spring MVC form tags

- Actually perform an HTTP POST, with the 'real' DELETE method hidden behind a request parameter, to be picked up by the *HiddenHttpMethodFilter*

```
<form:form method="delete">
  <p class="submit">
        <input type="submit" value="Delete Pet"/>
  </p>
</form:form>
```

# Server Side Code

- Controller code that handles HTTP DELETE

```
@RequestMapping(method = RequestMethod.DELETE)
public String deletePet(@PathVariable int ownerId,
                        @PathVariable int petId) {
    this.clinic.deletePet(petId);
    return "redirect:/owners/" + ownerId;
}
```

# Exception Handling

# Exception Handling

- In case of an exception processing the HTTP request, an exception of the type *RestClientException* will be thrown
  - This behavior can be changed by plugging in another *ResponseErrorHandler* implementation into the *RestTemplate*

# Object to/from HTTP Message Conversion

# Built-in Converter Classes

- Objects passed to and returned from these methods are converted to and from HTTP messages by *HttpMessageConverter* instances.
- Converters for the main mime types are registered by default, but you can also write your own converter and register it via the *messageConverters()* bean property.
- The default converter instances registered with the template are
  - ByteArrayHttpMessageConverter
  - StringHttpMessageConverter
  - FormHttpMessageConverter
  - SourceHttpMessageConverter

# Learn with Passion!
# JPassion.com