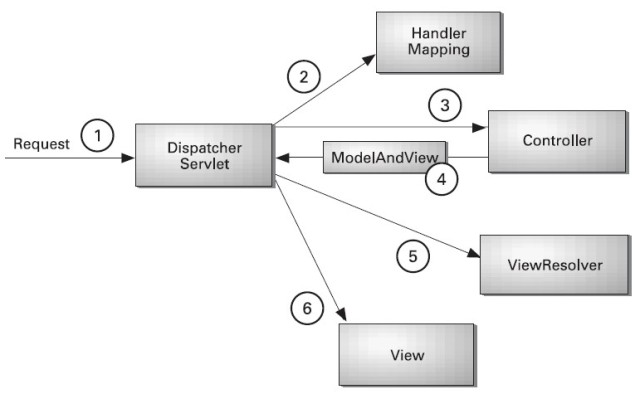
Spring MVC Basics:

The Spring web MVC framework provides model-view-controller architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

* The **Model** encapsulates the application data and in general they will consist of POJO.
* The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The **Controller** is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

Understanding the flow of Spring Web MVC



**21.4 Handler mappings**

In previous versions of Spring, users were required to define one or more HandlerMapping beans in the web application context to map incoming web requests to appropriate handlers. With the introduction of annotated controllers, you generally don’t need to do that because the RequestMappingHandlerMapping automatically looks for @RequestMapping annotations on all @Controller beans. However, do keep in mind that all HandlerMapping classes extending fromAbstractHandlerMapping have the following properties that you can use to customize their behavior:

* interceptors List of interceptors to use. HandlerInterceptors are discussed in [Section 21.4.1, “Intercepting requests with a HandlerInterceptor”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-handlermapping-interceptor).
* defaultHandler Default handler to use, when this handler mapping does not result in a matching handler.
* order Based on the value of the order property (see the org.springframework.core.Ordered interface), Spring sorts all handler mappings available in the context and applies the first matching handler.
* alwaysUseFullPath If true , Spring uses the full path within the current Servlet context to find an appropriate handler. If false (the default), the path within the current Servlet mapping is used. For example, if a Servlet is mapped using /testing/\* and the alwaysUseFullPath property is set to true,/testing/viewPage.html is used, whereas if the property is set to false, /viewPage.html is used.
* urlDecode Defaults to true, as of Spring 2.5. If you prefer to compare encoded paths, set this flag to false. However, the HttpServletRequest always exposes the Servlet path in decoded form. Be aware that the Servlet path will not match when compared with encoded paths.

The following example shows how to configure an interceptor:

<beans>

<bean id="handlerMapping" class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerMapping">

<property name="interceptors">

<bean class="example.MyInterceptor"/>

</property>

</bean>

<beans>

### Resolving views with the ViewResolver interface

As discussed in [Section 21.3, “Implementing Controllers”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-controller), all handler methods in the Spring Web MVC controllers must resolve to a logical view name, either explicitly (e.g., by returning a String, View, or ModelAndView) or implicitly (i.e., based on conventions). Views in Spring are addressed by a logical view name and are resolved by a view resolver. Spring comes with quite a few view resolvers. This table lists most of them; a couple of examples follow.

| **ViewResolver** | **Description** |
| --- | --- |
| AbstractCachingViewResolver | Abstract view resolver that caches views. Often views need preparation before they can be used; extending this view resolver provides caching. |
| XmlViewResolver | Implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring’s XML bean factories. The default configuration file is /WEB-INF/views.xml. |
| ResourceBundleViewResolver | Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name. Typically you define the bundle in a properties file, located in the classpath. The default file name isviews.properties. |
| UrlBasedViewResolver | Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings. |
| InternalResourceViewResolver | Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..). See the UrlBasedViewResolver javadocs for details. |
| VelocityViewResolver /FreeMarkerViewResolver | Convenient subclass of UrlBasedViewResolver that supports VelocityView (in effect, Velocity templates) orFreeMarkerView ,respectively, and custom subclasses of them. |
| ContentNegotiatingViewResolver | Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept header. See [Section 21.5.4, “ContentNegotiatingViewResolver”](http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html#mvc-multiple-representations). |

**Table 21.3. View resolvers**

As an example, with JSP as a view technology, you can use the UrlBasedViewResolver. This view resolver translates a view name to a URL and hands the request over to the RequestDispatcher to render the view.

<bean id="viewResolver"

class="org.springframework.web.servlet.view.UrlBasedViewResolver">

<property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>

<property name="prefix" value="/WEB-INF/jsp/"/>

<property name="suffix" value=".jsp"/>

</bean>

When returning test as a logical view name, this view resolver forwards the request to the RequestDispatcher that will send the request to/WEB-INF/jsp/test.jsp.

When you combine different view technologies in a web application, you can use the ResourceBundleViewResolver:

<bean id="viewResolver"

class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

<property name="basename" value="views"/>

<property name="defaultParentView" value="parentView"/>

</bean>

The ResourceBundleViewResolver inspects the ResourceBundle identified by the basename, and for each view it is supposed to resolve, it uses the value of the property [viewname].(class) as the view class and the value of the property [viewname].url as the view url. Examples can be found in the next chapter which covers view technologies. As you can see, you can identify a parent view, from which all views in the properties file "extend".

* @RestController (I know it is the same than @Controller + @ResponseBody)

ResponseEntity is meant to represent the entire HTTP response. You can control anything that goes into it: status code, headers, and body.

@ResponseBody is a marker for the HTTP response body and @ResponseStatus declares the status code of the HTTP response.

@ResponseStatus isn't very flexible. It marks the entire method so you have to be sure that your handler method will always behave the same way. And you still can't set the headers. You'd need the HttpServletResponse or a HttpHeaders parameter.

Basically, ResponseEntity lets you do more.

Example:

@RequestMapping(value="/{id}", method=RequestMethod.GET)

public ResponseEntity<Person> getPerson(@PathVariable Integer id){

logger.info("PersonRestResponseEntityController - getPerson - id: {}", id);

Person person = personMapRepository.findPerson(id);

return new ResponseEntity<>(person, HttpStatus.FOUND);

}

--===========-------------------

Spring @Profile allow developers to register beans by condition. For example, register beans based on what operating system (Windows, \*nix) your application is running, or load a database properties file based on the application running in development, test, staging or production environment.

In this tutorial, we will show you a Spring @Profile application, which does the following stuff :

1. Create two profiles – dev and live
2. If profile “dev” is enabled, return a simple cache manager – ConcurrentMapCacheManager
3. If profile “live” is enabled, return an advanced cache manager – EhCacheCacheManager.

----------------------99999999999999&&&&&&&&===================

**@ModelAttribute** has two usage scenarios in controllers. When you place it on a method parameter, @ModelAttribute maps a model attribute to the specific, annotated method parameter (see the processSubmit() method below). This is how the controller gets a reference to the object holding the data entered in the form.

You can also use @ModelAttribute at the method level to provide *reference data* for the model (see the getAllAddresses () method in the following example). For this usage the method signature can contain the same types as documented previously for the @RequestMapping annotation.

|  |  |
| --- | --- |
| [Note] | **Note** |
| @ModelAttribute annotated methods are executed *before* the chosen @RequestMapping annotated handler method. They effectively pre-populate the implicit model with specific attributes, often loaded from a database. Such an attribute can then already be accessed through @ModelAttribute annotated handler method parameters in the chosen handler method, potentially with binding and validation applied to it. |

* **Method:** If you need the model for a particular controller to be always populated with certain attributes the method level @ModelAttribute makes more sense.
* **Parameter:** Use it on a parameter when you want to bind data from the request and add it to the model implicitly.

Let me explain you further with the help of some examples.  
**Example 1:**

[?](http://thespringthing.blogspot.in/2010/11/how-does-modelattribute-work.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | @Controller  public class MyController {        @ModelAttribute("myobject")      public MyObject getInitializedMyObject() {          return myService.getInitializedObject();      }        @RequestMapping(value="/handle.htm", method=RequestMethod.GET)      public ModelAndView handleRequest() {          return new ModelAndView("myView");      }   } |

In this example, the value returned by getInitializedMyObject is added to the Model. The View would be able to retrieve this object using the key "myobject" from the request attributes.  
  
**Example 2 :**

[?](http://thespringthing.blogspot.in/2010/11/how-does-modelattribute-work.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | @Controller  public class MyController {        @ModelAttribute("myobject")      public MyObject getInitializeMyObject() {          return myService.getInitializedObject();      }        @RequestMapping(value="/handle.htm", method=RequestMethod.GET)      public ModelAndView handleRequest(@ModelAttribute("myobject") MyObject myObject) {          myObject.setValue("test");          return new ModelAndView("myView");      }    } |

In this case, the getInitializeMyObject is executed first and the result is stored in a temporary map. This value is then passed as a parameter to the handler method. And finally myObject is added to the model for the views.  
  
**Example 3 :**

[?](http://thespringthing.blogspot.in/2010/11/how-does-modelattribute-work.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | @Controller  @SessionAttributes("myobject")  public class MyController {        @RequestMapping(value="/handle.htm", method=RequestMethod.GET)      public ModelAndView handleRequest(@ModelAttribute("myobject") MyObject myObject) {          myObject.setValue("test");          return new ModelAndView("myView");      }    } |

In this case, Spring searches for "myobject" in the session and pass it as the parameter to the handler method. If "myobject" is not found in the session, then HttpSessionRequiredException is raised.  
  
  
**Example 4 :**

[?](http://thespringthing.blogspot.in/2010/11/how-does-modelattribute-work.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Controller  public class MyController {        @RequestMapping(value="/handle.htm", method=RequestMethod.GET)      public ModelAndView handleRequest(@ModelAttribute("myobject") MyObject myObject) {          myObject.setValue("test");          return new ModelAndView("myView");      }    } |

In this case, a new instance of MyObject is created and then passed to handler method. If MyObject is an interface or an abstract class, then a BeanInstantiationException is raised.

------------------------(((((((((((((((())))))))))))))))\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

***@SessionAttributes*** doesn't fully replaces the traditional *HttpServlet* session management. Use it if two or more Controller methods need to communicate some data. But, using this we can only achieve communication within single controller class. You do not use to read and write from and to the session explicitly if you are using *@SessionAttributes*. Usage of *@SessionAttributes* is suggested only for short lived communications. If you need to store long term data in session, it is suggested to use*session.setAttribute* and *session.getAttribute* explicitly, instead of *@SessionAttributes*. For more information

#### Binding request parameters to method parameters with @RequestParam

Use the @RequestParam annotation to bind request parameters to a method parameter in your controller.

**public** String setupForm(@RequestParam("petId")

**@Resource, @Autowired and @Inject**

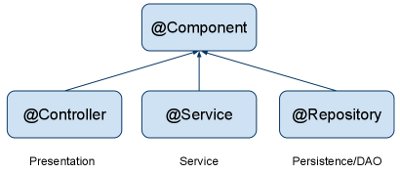
The main difference is that, @Autowired and @Inject works similar for 100% without any differentiation.These two annotations using AutowiredAnnotationBeanPostProcessor to inject dependencies. But,@Resource uses CommonAnnotationBeanPostProcessor to inject dependencies and there is difference in the order of checking.

**@Autowired and @Inject**

1. Matches by Type
2. Restricts by Qualifiers
3. Matches by Name

**@Resource**

1. Matches by Name
2. Matches by Type.3.Restricts by Qualifiers (ignored if match is found by name)

@Component is a generic stereotype for any Spring-managed component. @Repository, @Service, and @Controller are specializations of @Component for more specific use cases, for example, in the persistence, service, and presentation layers, respectively.   
  


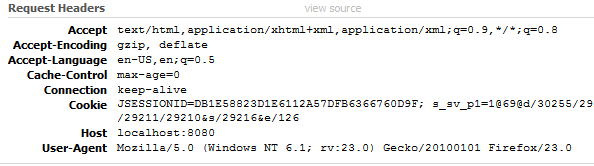
------------------------------))))))))))))))))))))



* @PathVariable annotated parameters for access to URI template variables. See [the section called “URI Template Patterns”](http://docs.spring.io/spring-framework/docs/3.2.0.M2/reference/html/mvc.html#mvc-ann-requestmapping-uri-templates).
* @MatrixVariable annotated parameters for access to name-value pairs located in URI path segments. See [the section called “Matrix Variables”](http://docs.spring.io/spring-framework/docs/3.2.0.M2/reference/html/mvc.html#mvc-ann-matrix-variables).
* @RequestParam annotated parameters for access to specific Servlet request parameters. Parameter values are converted to the declared method argument type. See [the section called “Binding request parameters to method parameters with @RequestParam”](http://docs.spring.io/spring-framework/docs/3.2.0.M2/reference/html/mvc.html#mvc-ann-requestparam).
* @RequestHeader annotated parameters for access to specific Servlet request HTTP headers. Parameter values are converted to the declared method argument type.
* @RequestBody annotated parameters for access to the HTTP request body. Parameter values are converted to the declared method argument type using HttpMessageConverters. See [the section called “Mapping the request body with the @RequestBody annotation”](http://docs.spring.io/spring-framework/docs/3.2.0.M2/reference/html/mvc.html#mvc-ann-requestbody).
* @RequestPart annotated parameters for access to the content of a "multipart/form-data" request part. See [Section 17.10.5, “Handling a file upload request from programmatic clients”](http://docs.spring.io/spring-framework/docs/3.2.0.M2/reference/html/mvc.html#mvc-multipart-forms-non-browsers) and [Section 17.10, “Spring's multipart (file upload) support”](http://docs.spring.io/spring-framework/docs/3.2.0.M2/reference/html/mvc.html#mvc-multipart).
* HttpEntity<?> parameters for access to the Servlet request HTTP headers and contents. The request stream will be converted to the entity body using HttpMessageConverters. See [the section called “Using HttpEntity<?>”](http://docs.spring.io/spring-framework/docs/3.2.0.M2/reference/html/mvc.html#mvc-ann-httpentity).
* java.util.Map / org.springframework.ui.Model / org.springframework.ui.ModelMap for enriching the implicit model that is exposed to the web view.

-----------------)))))))))))

[Spring MVC](http://www.javabeat.net/2013/10/spring-mvc-example/) provides an annotation **@RequestHeader** for facilitating use to get the header details easily in our controller class. This annotation would bind the header details with the method arguments, and it can be used inside the methods.

Look at the below picture for the details of header.

|  |
| --- |
| @RequestMapping(value = "/example", method = RequestMethod.GET) |

|  |  |
| --- | --- |
| 12 | public String  getHello(@RequestHeader ("host") String hostName, |

|  |  |
| --- | --- |
| 13 | @RequestHeader ("Accept") String acceptType, |

|  |  |
| --- | --- |
| 14 | @RequestHeader ("Accept-Language") String acceptLang, |

|  |  |
| --- | --- |
| 15 | @RequestHeader ("Accept-Encoding") String acceptEnc, |

|  |  |
| --- | --- |
| 16 | @RequestHeader ("Cache-Control") String cacheCon, |

|  |  |
| --- | --- |
| 17 | @RequestHeader ("Cookie") String cookie, |

|  |  |
| --- | --- |
| 18 | @RequestHeader ("User-Agent") String userAgent) |

|  |  |
| --- | --- |
| 19 | { |

|  |  |
| --- | --- |
| 27 | return "example"; |

|  |  |
| --- | --- |
| 28 | } |
|  |  |

# --------------------(((((((((((((((

**Format of an HTTP Request**  
  
It has three main components, which are:-

* **HTTP Request Method, URI, and Protocol Version** - this should always be the first line of an HTTP Request. As it's quite evident from the name itself, it contains the HTTP Request method being used for that particular request, the URI, and the HTTP protocol name with the version being used. It may look like 'GET /servlet/jspName.jsp HTTP/1.1' where the request method being used is 'GET', the URI is '/servlet/jspName.jsp', and the protocol (with version) is 'HTTP/1.1'.
* **HTTP Request Headers** - this section of an HTTP Request contains the request headers, which are used to communicate information about the client environment. Few of these headers are: Content-Type, User-Agent, Accept-Encoding,Content-Length, Accept-Language, Host, etc. Very obvious to understand what info do these headers carry, isn't it? The names are quite self-explanatory.
* **HTTP Request Body** - this part contains the actual request being sent to the HTTP Server. The HTTP Request Header and Body are separated by a blank line (CRLF sequence, where CR means Carriage Return and LF means Line Feed). This blank line is a mandatory part of a valid HTTP Request.

**Format of an HTTP Response**  
  
Similar to an HTTP Request, an HTTP Response also has three main components, which are:-

* **Protocol/Version, Status Code, and its Description** - the very first line of a valid HTTP Response is consists of the protocol name, it's version, status code of the request, and a short description of the status code. A status code of 200means the processing of request was successful and the description in this case will be 'OK'. Similarly, a status code of '404' means the file requested was not found at the HTTP Server at the expected location and the description in this case is 'File Not Found'.
* **HTTP Response Headers** - similar to HTTP Request Headers, HTTP Response Headers also contain useful information. The only difference is that HTTP Request Headers contain information about the environment of the client machine whereas HTTP Response Headers contain information about the environment of the server machine. This is easy to understand as HTTP Requests are formed at the client machine whereas HTTP Responses are formed at the server machine. Few of these HTTP Response headers are: Server, Content-Type, Last-Modified, Content-Length, etc.
* **HTTP Response Body** - this the actual response which is rendered in the client window (the browser window). The content of the body will be HTML code. Similar to HTTP Request, in this case also the Body and the Headers components are separated by a mandatory blank line (CRLF sequence).

--------))))))))))))---------------

# [How to include js and CSS in JSP with spring MVC](http://stackoverflow.com/questions/26276603/how-to-include-js-and-css-in-jsp-with-spring-mvc)

First you need to declare your resources in dispatcher-servlet file like this :

<mvc:resources mapping="/resources/\*\*" location="/resources/folder/" />

Any request with url mapping /resources/\*\* will directly look for /resources/folder/.

Now in jsp file you need to include your css file like this :

<link href="<c:url value="/resources/css/main.css" />" rel="stylesheet">

Similarly you can include js files.

Hope this solves your problem.

# what different between InternalResourceViewResolver vs UrlBasedViewResolver

1. InternalResourceViewResolver is a convenient subclass of UrlBasedViewResolver.

The **java**Doc describes some added properties in InternalResourceViewResolver that might be useful in some situations:

Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (i.e. Servlets and JSPs) and subclasses such as JstlView.

AlwaysInclude: Controls whether either a forward or include is done.

ExposeContextBeansAsAttributes: Allows all beans in context to be available as request attributes, which means they can be referenced from the EL in JSP.

ExposedContextBeanNames: If non-null, specifies the list of beans that will be exposed, as opposed to all of them.

Source from *spring* forum : *spring* Q&A forum

[JstlView](http://static.springframework.org/spring/docs/2.5.x/api/org/springframework/web/servlet/view/JstlView.html)has a **exposeContextBeansAsAttributes** property you can use to expose all your Spring beans to JSTL? I didn't. To configure it, you configure your viewResolver as follows:

<bean id="viewResolver"

class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>

<property name="exposeContextBeansAsAttributes" value="true"/>

<property name="prefix" value="/"/>

<property name="suffix" value=".jsp"/>

</bean>

After doing this, any Spring bean can get referenced in JSTL with:

${beanId.getterMethodWithoutTheGetPrefix}

Second :

<!-- defined view resolver to resolve Controller class HelloWorldController to helloworld.jsp -->

   <bean id="viewResolver"

          class="org.springframework.web.servlet.view.InternalResourceViewResolver"

          p:prefix="/WEB-INF/jsp/"

          p:suffix=".jsp">

<!-- exposing Controller bean helloWorldController to the view  -->

          <property name="exposedContextBeanNames">

            <list>

             <value>helloWorldController</value>

            </list>

          </property>

    </bean>

<!-- Explicit mapping to Controller bean helloWorldController which references Service bean thru its property

       from the applicationContext.xml -->

    <bean id= "helloWorldController"

      class="controller.HelloWorldController" p:helloWorldService-ref="helloWorldService"/>

<!-- ${beanName.beanProperty}-->

     ${helloWorldController.helloWorldService.message}

### setAlwaysInclude

public void **setAlwaysInclude**(boolean alwaysInclude)

Specify whether to always include the view rather than forward to it.

Default is "false". Switch this flag on to enforce the use of a Servlet include, even if a forward would be possible.

**See Also:**

[InternalResourceView.setAlwaysInclude(boolean)](http://docs.spring.io/spring/docs/3.0.x/api/org/springframework/web/servlet/view/InternalResourceView.html#setAlwaysInclude(boolean))

### setExposeContextBeansAsAttributes

public void **setExposeContextBeansAsAttributes**(boolean exposeContextBeansAsAttributes)

Set whether to make all Spring beans in the application context accessible as request attributes, through lazy checking once an attribute gets accessed.

This will make all such beans accessible in plain ${...} expressions in a JSP 2.0 page, as well as in JSTL's c:out value expressions.

Default is "false".

**See Also:**

[InternalResourceView.setExposeContextBeansAsAttributes(boolean)](http://docs.spring.io/spring/docs/3.0.x/api/org/springframework/web/servlet/view/InternalResourceView.html#setExposeContextBeansAsAttributes(boolean))

### setExposedContextBeanNames

public void **setExposedContextBeanNames**([String](http://java.sun.com/javase/6/docs/api/java/lang/String.html?is-external=true)[] exposedContextBeanNames)

Specify the names of beans in the context which are supposed to be exposed. If this is non-null, only the specified beans are eligible for exposure as attributes.

Exception Handling IN Spring MVC:

**What to Use When?**

As usual, Spring likes to offer you choice, so what should you do? Here are some rules of thumb.  
However if you have a preference for XML configuration or Annotations, that’s fine too.

* For exceptions you write, consider adding @ResponseStatus to them.
* For all other exceptions implement an @ExceptionHandler method on a@ControllerAdvice class or use an instance of SimpleMappingExceptionResolver. You may well have SimpleMappingExceptionResolver configured for your application already, in which case it may be easier to add new exception classes to it than implement a@ControllerAdvice.
* For Controller specific exception handling add @ExceptionHandler methods to your controller.
* **Warning:** Be careful mixing too many of these options in the same application. If the same exception can be handed in more than one way, you may not get the behavior you wanted.@ExceptionHandler methods on the Controller are always selected before those on any@ControllerAdvice instance. It is *undefined* what order controller-advices are processed.

## Controller Based Exception Handling

### Using @ExceptionHandler

You can add extra (@ExceptionHandler) methods to any controller to specifically handle exceptions  
thrown by request handling (@RequestMapping) methods in the same controller. Such methods can:

1. Handle exceptions without the @ResponseStatus annotation (typically predefined exceptions  
   that you didn’t write)
2. Redirect the user to a dedicated error view
3. Build a totally custom error response

The following controller demonstrates these three options:

@Controller

public class ExceptionHandlingController {

// @RequestHandler methods

...

// Exception handling methods

// Convert a predefined exception to an HTTP Status code

@ResponseStatus(value=HttpStatus.CONFLICT, reason="Data integrity violation") // 409

@ExceptionHandler(DataIntegrityViolationException.class)

public void conflict() {

// Nothing to do

}

// Specify the name of a specific view that will be used to display the error:

@ExceptionHandler({SQLException.class,DataAccessException.class})

public String databaseError() {

// Nothing to do. Returns the logical view name of an error page, passed to

// the view-resolver(s) in usual way.

// Note that the exception is \_not\_ available to this view (it is not added to

// the model) but see "Extending ExceptionHandlerExceptionResolver" below.

return "databaseError";

}

// Total control - setup a model and return the view name yourself. Or consider

// subclassing ExceptionHandlerExceptionResolver (see below).

@ExceptionHandler(Exception.class)

public ModelAndView handleError(HttpServletRequest req, Exception exception) {

logger.error("Request: " + req.getRequestURL() + " raised " + exception);

ModelAndView mav = new ModelAndView();

mav.addObject("exception", exception);

mav.addObject("url", req.getRequestURL());

mav.setViewName("error");

return mav;

}

}

In any of these methods you might choose to do additional processing - the most common example is to log the  
exception.

Handler methods have flexible signatures so you can pass in obvious servlet-related objects such  
as HttpServletRequest, HttpServletResponse, HttpSession and/or Principle. **Important Note:** the  
Model may **not** be a parameter of any @ExceptionHandler method. Instead, setup a model inside the method  
using a ModelAndView as shown by handleError() above.

### Exceptions and Views

Be careful when adding exceptions to the model. Your users do not want to see  
web-pages containing Java exception details and stack-traces. However, it can be useful to put exception  
details in the page source as a comment, to assist your support people. If using JSP, you could  
do something like this to output the exception and the corresponding stack-trace (using a hidden  
<div> is another option).

<h1>Error Page</h1>

<p>Application has encountered an error. Please contact support on ...</p>

<!--

Failed URL: ${url}

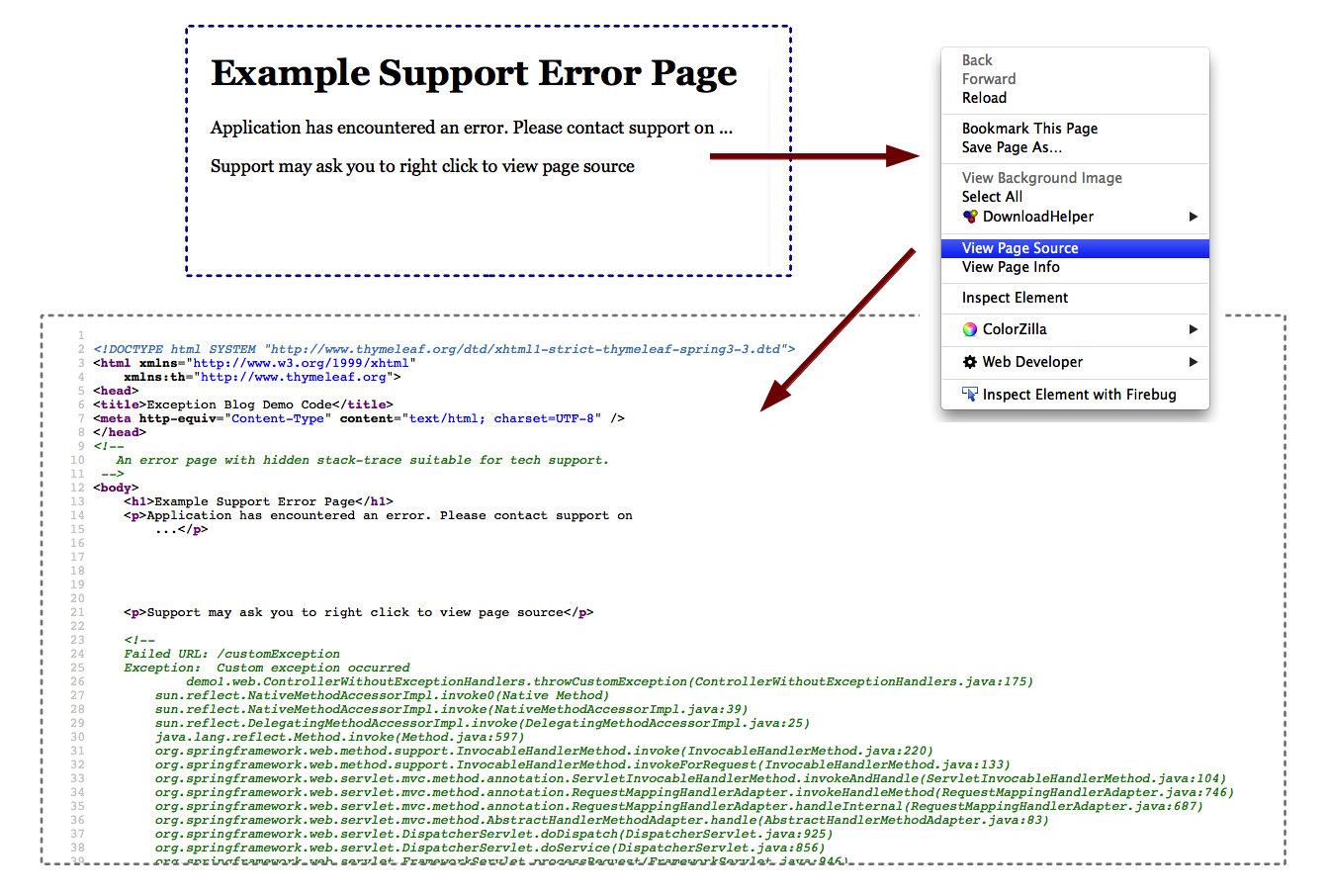
Exception: ${exception.message}

<c:forEach items="${exception.stackTrace}" var="ste"> ${ste}

</c:forEach>

-->

For the Thymeleaf equivalent see  
[support.html](https://github.com/paulc4/mvc-exceptions/blob/master/src/main/resources/templates/support.html)  
in the demo application. The result looks like this.



## Global Exception Handling

### Using @ControllerAdvice Classes

A controller advice allows you to use exactly the same exception handling techniques but apply them  
across the whole application, not just to an individual controller. You can think of them as an annotation  
driven interceptor.

Any class annotated with @ControllerAdvice becomes a controller-advice and three types of method  
are supported:

* Exception handling methods annotated with @ExceptionHandler.
* Model enhancement methods (for adding additional data to the model) annotated with  
  @ModelAttribute. Note that these attributes are not available to the exception handling views.
* Binder initialization methods (used for configuring form-handling) annotated with  
  @InitBinder.

We are only going to look at exception handling - see the online manual for more on  
@ControllerAdvice methods.

Any of the exception handlers you saw above can be defined on a controller-advice class - but now they  
apply to exceptions thrown from any controller. Here is a simple example:

@ControllerAdvice

class GlobalControllerExceptionHandler {

@ResponseStatus(HttpStatus.CONFLICT) // 409

@ExceptionHandler(DataIntegrityViolationException.class)

public void handleConflict() {

// Nothing to do

}

}

If you want to have a default handler for any exception, there is a slight wrinkle. You need to ensure  
annotated exceptions are handled by the framework. The code looks like this:

@ControllerAdvice

class GlobalDefaultExceptionHandler {

public static final String DEFAULT\_ERROR\_VIEW = "error";

@ExceptionHandler(value = Exception.class)

public ModelAndView defaultErrorHandler(HttpServletRequest req, Exception e) throws Exception {

// If the exception is annotated with @ResponseStatus rethrow it and let

// the framework handle it - like the OrderNotFoundException example

// at the start of this post.

// AnnotationUtils is a Spring Framework utility class.

if (AnnotationUtils.findAnnotation(e.getClass(), ResponseStatus.class) != null)

throw e;

// Otherwise setup and send the user to a default error-view.

ModelAndView mav = new ModelAndView();

mav.addObject("exception", e);

mav.addObject("url", req.getRequestURL());

mav.setViewName(DEFAULT\_ERROR\_VIEW);

return mav;

}

}

## Going Deeper

### HandlerExceptionResolver

Any Spring bean declared in the DispatcherServlet’s application context that implements  
HandlerExceptionResolver will be used to intercept and process any exception raised  
in the MVC system and not handled by a Controller. The interface looks like this:

public interface HandlerExceptionResolver {

ModelAndView resolveException(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex);

}

The handler refers to the controller that generated the exception (remember that  
@Controller instances are only one type of handler supported by Spring MVC.  
For example: HttpInvokerExporter and the WebFlow Executor are also types of handler).

Behind the scenes, MVC creates three such resolvers by default. It is these resolvers that implement the  
behaviours discussed above:

* ExceptionHandlerExceptionResolver matches uncaught exceptions against for  
  suitable @ExceptionHandler methods on both the handler (controller) and on any controller-advices.
* ResponseStatusExceptionResolver looks for uncaught exceptions  
  annotated by @ResponseStatus (as described in Section 1)
* DefaultHandlerExceptionResolver converts standard Spring exceptions and converts them to HTTP Status Codes (I have not mentioned this above as it is internal to Spring MVC).

These are chained and processed in the order listed (internally Spring creates a dedicated bean - the  
HandlerExceptionResolverComposite to do this).

Notice that the method signature of resolveException does not include the Model. This is why  
@ExceptionHandler methods cannot be injected with the model.

You can, if you wish, implement your own HandlerExceptionResolver to setup your own custom  
exception handling system. Handlers typically implement Spring’s Ordered interface so you can define the  
order that the handlers run in.

### SimpleMappingExceptionResolver

Spring has long provided a simple but convenient implementation ofHandlerExceptionResolver  
that you may well find being used in your appication already - theSimpleMappingExceptionResolver.  
It provides options to:

* Map exception class names to view names - just specify the classname, no package needed.
* Specify a default (fallback) error page for any exception not handled anywhere else
* Log a message (this is not enabled by default).
* Set the name of the exception attribute to add to the Model so it can be used inside a View  
  (such as a JSP). By default this attribute is named exception. Set to null to disable. Remember  
  that views returned from @ExceptionHandler methods do not have access to the exception but views  
  defined to SimpleMappingExceptionResolver do.

Here is a typical configuration using XML:

<bean id="simpleMappingExceptionResolver"

class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">

<property name="exceptionMappings">

<map>

<entry key="DatabaseException" value="databaseError"/>

<entry key="InvalidCreditCardException" value="creditCardError"/>

</map>

</property>

<!-- See note below on how this interacts with Spring Boot -->

<property name="defaultErrorView" value="error"/>

<property name="exceptionAttribute" value="ex"/>

<!-- Name of logger to use to log exceptions. Unset by default, so logging disabled -->

<property name="warnLogCategory" value="example.MvcLogger"/>

</bean>

Or using Java Configuration:

@Configuration

@EnableWebMvc // Optionally setup Spring MVC defaults if you aren't doing so elsewhere

public class MvcConfiguration extends WebMvcConfigurerAdapter {

@Bean(name="simpleMappingExceptionResolver")

public SimpleMappingExceptionResolver createSimpleMappingExceptionResolver() {

SimpleMappingExceptionResolver r =

new SimpleMappingExceptionResolver();

Properties mappings = new Properties();

mappings.setProperty("DatabaseException", "databaseError");

mappings.setProperty("InvalidCreditCardException", "creditCardError");

r.setExceptionMappings(mappings); // None by default

r.setDefaultErrorView("error"); // No default

r.setExceptionAttribute("ex"); // Default is "exception"

r.setWarnLogCategory("example.MvcLogger"); // No default

return r;

}

...

}

The defaultErrorView property is especially useful as it ensures any uncaught exception generates  
a suitable application defined error page. (The default for most application servers is to display a Java  
stack-trace - something your users should never see).

### Extending SimpleMappingExceptionResolver

It is quite common to extend SimpleMappingExceptionResolver for several reasons:

* Use the constructor to set properties directly - for example to enable exception logging and set the  
  logger to use
* Override the default log message by overriding buildLogMessage. The default implementation  
  always returns this fixed text:

*Handler execution resulted in exception*

* To make additional information available to the error view by overridingdoResolveException

For example:

public class MyMappingExceptionResolver extends SimpleMappingExceptionResolver {

public MyMappingExceptionResolver() {

// Enable logging by providing the name of the logger to use

setWarnLogCategory(MyMappingExceptionResolver.class.getName());

}

@Override

public String buildLogMessage(Exception e, HttpServletRequest req) {

return "MVC exception: " + e.getLocalizedMessage();

}

@Override

protected ModelAndView doResolveException(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception exception) {

// Call super method to get the ModelAndView

ModelAndView mav = super.doResolveException(request, response, handler, exception);

// Make the full URL available to the view - note ModelAndView uses addObject()

// but Model uses addAttribute(). They work the same.

mav.addObject("url", request.getRequestURL());

return mav;

}

}

This code is in the demo application as  
[ExampleSimpleMappingExceptionResolver](https://github.com/paulc4/mvc-exceptions/blob/master/src/main/java/demo1/web/ExampleSimpleMappingExceptionResolver.java)

### Extending ExceptionHandlerExceptionResolver

It is also possible to extend ExceptionHandlerExceptionResolver and override its  
doResolveHandlerMethodException method in the same way. It has almost the same signature  
(it just takes the new HandlerMethod instead of a Handler).

To make sure it gets used, also set the inherited order property (for example in the constructor of  
your new class) to a value less than MAX\_INT so it runs before the default  
ExceptionHandlerExceptionResolver instance (it is easier to create your own handler instance than try to  
modify/replace the one created by Spring). See  
[ExampleExceptionHandlerExceptionResolver](http://github.com/paulc4/mvc-exceptions/blob/master/src/main/java/demo1/web/ExampleExceptionHandlerExceptionResolver.java)  
in the demo app for more.

### ****. Returning Status Codes via an Exception****

We will add a second method to the controller to demonstrate how to use an Exception to return a status code:

|  |  |
| --- | --- |
| 1  2  3  4  5 | @RequestMapping(value = "/exception", method = RequestMethod.GET)  @ResponseBody  public ResponseEntity sendViaException() {      throw new ForbiddenException();  } |

Upon receiving a GET request to “/exception“, Spring will throw a ForbiddenException. This is a custom exception that we will define in a separate class:

|  |  |
| --- | --- |
| 1  2 | @ResponseStatus(HttpStatus.FORBIDDEN)  public class ForbiddenException extends RuntimeException {} |

No code is required in this exception. All the work is done by the @ResponseStatus annotation. In this case, when the exception is thrown, the controller that threw it returns a response with the response code 403 (Forbidden). If necessary, you can also add a message in the annotation that will be returned along with the response. In this case, the class would look like this:

|  |  |
| --- | --- |
| 1  2 | @ResponseStatus(value = HttpStatus.FORBIDDEN, reason="To show an example of a custom message")  public class ForbiddenException extends RuntimeException {}  ========================= |

### 

### Errors and REST

RESTful GET requests may also generate exceptions and we have already seen how we can return standard HTTP  
Error response codes. However, what if you want to return information about the error? This is very easy to do.  
Firstly define an error class:

public class ErrorInfo {

public final String url;

public final String ex;

public ErrorInfo(String url, Exception ex) {

this.url = url;

this.ex = ex.getLocalizedMessage();

}

}

Now we can return an instance from a handler as the @ResponseBody like this:

@ResponseStatus(HttpStatus.BAD\_REQUEST)

@ExceptionHandler(MyBadDataException.class)

@ResponseBody ErrorInfo handleBadRequest(HttpServletRequest req, Exception ex) {

return new ErrorInfo(req.getRequestURL(), ex);

}

-------------------\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*------------------------------

Exception handlings:

===========

The DispatcherServlet handles resolving the exceptions using a [HandlerExceptionResolver](http://docs.spring.io/spring/docs/3.2.x/javadoc-api/org/springframework/web/servlet/HandlerExceptionResolver.html)instance. There are three main implementations of this class:

1. ExceptionHandlerExceptionResolver
2. ResponseStatusExceptionResolver
3. DefaultHandlerExceptionResolver

At context initialization, Spring will generate a ControllerAdviceBean for each @ControllerAdviceclass it detects. Each of those will then register a ExceptionHandlerMethodResolver with a Map of @ExceptionHandler handler methods. These will then be added to the ExceptionHandlerExceptionResolver instance.

The DispatcherServlet iterates through the HandlerExceptionResolver instances registered above (in order) until it can find one that handles the exception. In this case, we are interested in ExceptionHandlerExceptionResolver (which is first) which maps all @ExceptionHandler methods. In its getExceptionHandlerMethod method, it iterates through an entry set of Entry<ControllerAdviceBean, ExceptionHandlerMethodResolver> that it previously registered. If it finds an resolver for your Exception it will use it right away.

So the point here is: if you have a @ControllerAdvice with an @ExceptionHandler for Exceptionthat gets registered before another @ControllerAdvice class with an @ExceptionHandler for a more specific exception, like IOException, that first one will get called.

# [What is the difference between ResponseEntity<T> and @ResponseBody?](http://stackoverflow.com/questions/22725143/what-is-the-difference-between-responseentityt-and-responsebody)

ResponseEntity will give you some added flexibility in defining arbitrary HTTP response headers. See the 4th constructor here:

<http://docs.spring.io/spring/docs/3.0.x/api/org/springframework/http/ResponseEntity.html>

ResponseEntity(T body, MultiValueMap<String,String> headers, HttpStatus statusCode)

A List of possible HTTP response headers is available here:

<http://en.wikipedia.org/wiki/List_of_HTTP_header_fields#Responses>

Some commonly-used ones are Status, Content-Type and Cache-Control.

If you don't need that, using @ResponseBody will be a tiny bit more concise.

**BeanNameUrlHandlerMapping**

This is the default implementation used by the [DispatcherServlet](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/DispatcherServlet.html), along with [DefaultAnnotationHandlerMapping](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/mvc/annotation/DefaultAnnotationHandlerMapping.html). Alternatively, [SimpleUrlHandlerMapping](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/handler/SimpleUrlHandlerMapping.html) allows for customizing a handler mapping declaratively.

The mapping is from URL to bean name. Thus an incoming URL "/foo" would map to a handler named "/foo", or to "/foo /foo2" in case of multiple mappings to a single handler. Note: In XML definitions, you'll need to use an alias name="/foo" in the bean definition, as the XML id may not contain slashes.

**SimpleUrlHandlerMapping :**

Mappings to bean names can be set via the "mappings" property, in a form accepted by the java.util.Properties class, like as follows:  
/welcome.html=ticketController /show.html=ticketController   
The syntax is PATH=HANDLER\_BEAN\_NAME. If the path doesn't begin with a slash, one is prepended.

order Based on the value of the order property (see the org.springframework.core.Ordered interface), Spring sorts all handler mappings available in the context and applies the first matching handler.

**interface ControllerAdvice**

Indicates the annotated class assists a "Controller".

Serves as a specialization of [@Component](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/stereotype/Component.html), allowing for implementation classes to be autodetected through classpath scanning.

It is typically used to define [@ExceptionHandler](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ExceptionHandler.html), [@InitBinder](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/InitBinder.html), and [@ModelAttribute](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/ModelAttribute.html) methods that apply to all [@RequestMapping](http://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html) methods.

@InitBinder ::

1) Before, you had to resort to manually parsing the date:

public void webmethod(@RequestParam("date") String strDate) {

Date date = ... // manually parse the date

}

Now you can get the parsed date directly:

public void webmethod(@RequestParam("date") Date date) {

}

2) If your jsp page supplies a date on the form yyyy-MM-dd you can retrieve it as a Date object directly in your controller.

3) Spring tries against all registered editors to see if values can be converted into objects. You don't have to do anything in the object itself, that's the beauty of it.

@Controller

**public class** MyFormController {

**@InitBinder**

**protected void** initBinder(WebDataBinder binder) {

binder.addCustomFormatter(**new** DateFormatter(***"yyyy-MM-dd"***));

}

*// ...*

}bbv

Property Editors and Convertes :

In my mind, PropertyEditors are limited in scope - they help convert String to a type, and this string typically comes from UI, and so registering a PropertyEditor using @InitBinder and using WebDataBinder makes sense.

Converter on the other hand is more generic, it is intended for ANY conversion in the system - not just for UI related conversions(String to target type). For eg, Spring Integration uses a converter extensively for converting a message payload to a desired type.

I think for UI related flows PropertyEditors are still appropriate especially for the case where you need to do something custom for a specific command property. For other cases, I would take the recommendation from Spring reference and write a converter instead(for eg, to convert from a Long id to an entity say, as a sample)

Converters :

So let's say I want to use Converters because it is "the latest alternative". I would have to create **two** converters :

public class StringToCategory implements Converter<String, Category> {

@Override

public Category convert(String source) {

Category c = new Category(source);

return c;

}

}

public class CategoryToString implements Converter<Category, String> {

@Override

public String convert(Category source) {

return source.getName();

}

}

<bean id="conversionService"

class="org.springframework.context.support.ConversionServiceFactoryBean">

<property name="converters">

<set>

<bean class="somepackage.StringToCategory"/>

<bean class="somepackage.CategoryToString"/>

</set>

</property>

</bean>

### Validator

org.springframework.validation.Validator is an interface which has two methods   
**boolean supports(Class<?> clazz)**: Checks whether the instance of class passed as argument can be validated or not.   
**void validate(Object target, Errors errors)**: If *supports()*method returns true, then the

target object is validated. *Errors.rejectValue()* method registers the error message with a field name.

## View Resolvers:

[ResourceBundleViewResolver](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/view/ResourceBundleViewResolver.html) :

[ResourceBundleViewResolver](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/view/ResourceBundleViewResolver.html) if you need to apply different view resources per locale.

A [ViewResolver](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/servlet/ViewResolver.html) implementation that uses bean definitions in a [ResourceBundle](https://docs.oracle.com/javase/8/docs/api/java/util/ResourceBundle.html?is-external=true), specified by the bundle basename.

The bundle is typically defined in a properties file, located in the classpath. The default bundle basename is "views".

This ViewResolver supports localized view definitions, using the default support of [PropertyResourceBundle](https://docs.oracle.com/javase/8/docs/api/java/util/PropertyResourceBundle.html?is-external=true). For example, the basename "views" will be resolved as class path resources "views\_de\_AT.properties", "views\_de.properties", "views.properties" - for a given Locale "de\_AT".

# Spring File uploads:

Spring’s built-in multipart support handles file uploads in web applications. You enable

this multipart support with pluggable MultipartResolver objects, defined in the

org.springframework.web.multipart package. Spring provides one MultipartResolver

implementation for use with *Commons FileUpload* and another for use with Servlet 3.0 multipart request

parsing.

Spring provides file upload support using MultiPartResolver interface and provides two out-of-box implementations for that.

* **1. To use with Apache Commons**. Spring CommonsMultipartResolver is a MultipartResolver implementation for use with Apache Commons FileUpload. It requires apache commons-fileupload.jar to be present on classpath. It’s not specific to Servlet 3 environment but it works equally well with Servlet 3.x containers.

|  |
| --- |
| <dependency>      <groupId>commons-fileupload</groupId>      <artifactId>commons-fileupload</artifactId>      <version>1.3.1</version>  </dependency> |

* **2. To use with Servlet 3.0 multipart request.** Depends on which setup you are using[XML or JavaConfig].

For XML setup, you need to mark the DispatcherServlet with

a "multipart-config" section in web.xml.  
For Annoataion/JavaConfig setup, registering javax.servlet.MultipartConfigElement with DispatcherServlet. OR using @MultipartConfig on a custom servlet.

**CommonsMultipartResolver::**

**Basically we need to do following :**

* Create a Bean of Type CommonsMultipartResolver , specifying few properties related to file uploading.
* Include Apache Commons **commons-fileupload.jar** in classpath.
* Form which contains file upload functionality should specify enctype attribute with multipart content [**enctype=”multipart/form-data”**]
* To handle file input, input type must be ‘file’

Something like this:

|  |
| --- |
| <form method="POST" enctype="multipart/form-data" >  ....      <input type="file" id="file" />  ....      <input type="submit" value="Upload">  </form> |

<bean id="multipartResolver" class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

        <!-- one of the several properties available; the maximum file size in bytes -->

        <property name="maxUploadSizePerFile" value="5242880"/>

    </bean>

n order to activate Multipart support with Spring in Servlet 3.0 environment, you need to do following

**1.** Add StandardServletMultipartResolver Bean to your Spring Configuration.It’s a standard implementation of the MultipartResolver interface, based on the Servlet 3.0 javax.servlet.http.Part API.

**2.** **Enable MultiParsing in Servlet 3.0 environments. To do that, You have several choices to choose from.**

* **Choice A.** Set javax.servlet.MultipartConfigElement in programmatic Servlet registration. MultipartConfigElement is simply Java Class representation of an javax.servlet.annotation.MultipartConfig annotation value (as described in choice c). This post will focus specially on this choice.
* **Choice B.** If you are using XML based configuration, you can declare multipart-config section under servlet configuration in web.xml, as shown below:

|  |
| --- |
| <servlet>      <servlet-name>SpringDispatcher</servlet-name>      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>      <multipart-config>          <location>/tmp</location>          <max-file-size>5242880</max-file-size><!--5MB-->          <max-request-size>20971520</max-request-size><!--20MB-->          <file-size-threshold>0</file-size-threshold>      </multipart-config>  </servlet> |

* **Choice C.** You can create a custom Servlet and annotate it with javax.servlet.annotation.MultipartConfigannotation as shown below:

|  |
| --- |
| @WebServlet(name = "fileUploadServlet", urlPatterns = {"/upload"})  @MultipartConfig(location=/tmp,                   fileSizeThreshold=0,                   maxFileSize=5242880,       // 5 MB                   maxRequestSize=20971520)   // 20 MB  public class FileUploadServlet extends HttpServlet {        protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {          //handle file upload      } |

That being said, we will focus on Choice A in this example.

# Spring MVC – component-scan Vs annotation-config Vs annotation-driven

If you are a spring developer then you might have a small confusion about the annotation processing inside spring’s IOC container. **Spring MVC** framework provides different configuration elements that are helping or instructing the Spring container to effectively manage the beans and inject the beans when required. Some of the XML configurations that are most commonly seen in our spring configuration files are:

* **context:component-scan**
* **mvc:annotation-config**
* **context:annotation-driven**

The functionality of the above annotations are similar and there is little variations on how they are reacting on the specific scenarios. This tutorial highlights the important point about each element and when it is required for the application. If you have any queries on Spring Framework, please write it in the comments or post it in our [facebook page](http://facebook.com/javabeat.net" \t "_blank).

**also read:**

* [Spring and Hibernate Integration](http://www.javabeat.net/integrating-spring-framework-with-hibernate-orm-framework/)
* [Spring MVC Framework](http://www.javabeat.net/introduction-to-spring-mvc-web-framework-web-tier/)

## context:component-scan

This element has been introduced in Spring configuration from version 2.5. If you have worked with the previous versions of Spring, all the beans has to be manually configured in the XML files. There are no annotations supported in the Java beans. This will result in lot of XML code in the configuration files and every time developer has to update the XML file for configuring the new beans. **context:component-scan** element in the spring configuration file would eliminate the need for declaring all the beans in the XML files. Look at the below declaration in your spring configuration file:

<context:component-scan base-package="org.controller"/>

The above declaration in the spring application configuration file would scan the classes inside the specified package and create the beans instance. Note that it could create beans only if that class is annotated with correct annotations. The following are the annotations scanned by this element:

* **@Component**
* **@Repository**
* **@Service**
* **@Controller**

One advantage of this element is that it also resolve [@Autowired](http://www.javabeat.net/autowired-annotation-spring/) and [@Qualifier](http://www.javabeat.net/qualifier-annotation-spring/) annotations. Therefore if you declare **<context:component-scan>**, is not necessary anymore declare **<context:annotation-config>** too.

## mvc:annotation-driven

**mvc:annotation-driven** is used for enabling the Spring MVC components with its default configurations. If you dont include mvc:annotation-driven also your MVC application would work if you have used the context:component-scan for creating the beans or defined the beans in your XML file. But, mvc:annotation-driven does some extra job on configuring the special beans that would not have been configured if you are not using this element in your XML file.

This tag would registers the HandlerMapping and HandlerAdapter required to dispatch requests to your @Controllers. In addition, it also applies some defaults based on what is present in your classpath. Such defaults are:

* Using the Spring 3 Type ConversionService as a simpler and more robust alternative to JavaBeans PropertyEditors
* Support for formatting Number fields with @NumberFormat
* Support for formatting Date, Calendar, and Joda Time fields with @DateTimeFormat, if Joda Time is on the classpath
* Support for validating @Controller inputs with @Valid, if a JSR-303 Provider is on the classpath
* Support for reading and writing XML, if JAXB is on the classpath
* Support for reading and writing JSON, if Jackson is on the classpath

## context:annotation-config

context:annotation-config is used for activating annotations in beans already registered in the application context (no matter whether they were defined with XML or by package scanning). That means it will resolve @Autowired and @Qualifier annotations for the beans which are already created and stored in the spring container.

**context:component-scan** can also do the same job, but context:component-scan will also scan the packages for registering the beans to application context. context:annotation-config will not search for the beans registration, this will only activate the already registered beans in the context.

Please read this [**link**](http://stackoverflow.com/questions/7414794/difference-between-contextannotation-config-vs-contextcomponent-scan) for more explanation with examples.

If you are looking for any support on Spring MVC applications, please post it in the comments section. We are happy to help you and resolve the technical issues. Hap

===================------------------------==================================

# Content Negotiation using Views

Spring MVC has long supported multiple view resolvers, and goes to each in turn to find a view. Although the order that view resolvers are consulted can be specified, Spring MVC always picks the first view offered.  The ’‘Content Negotiating View Resolver’’ (CNVR) negotiates between all the view resolvers to find the best match for the format desired - this is our “smart” view resolver.

The ContentNegotiationManager created by either setup is an implementation of ContentNegotationStrategy that implements the **PPA Strategy** (path extension, then parameter, then Accept header) described above.

<!-- Setup a simple strategy:

1. Take all the defaults.

2. Return XML by default when not sure. -->

<bean id="contentNegotiationManager" class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">

<property name="defaultContentType" value="application/xml" /> </bean>

<!-- Make this available across all of Spring MVC -->

<mvc:annotation-driven content-negotiation-manager="contentNegotiationManager" />

The Github demo project uses 2 sets of Spring profiles either separate or combined. The separate profile defines all view resolvers as top-level beans and lets the CNVR scan the context to find them (as discussed in the previous section). In the combined profile the view resolvers are defined explicitly, not as Spring beans and passed to the CNVR via its viewResolvers property (as shown in this section).

Combined profile :

<bean id="cnManager" class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean">

<property name="ignoreAcceptHeader" value="true"/>

<property name="defaultContentType" value="text/html" />

</bean>

<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">

<property name="contentNegotiationManager" ref="cnManager"/>

<!-- Define the view resolvers explicitly -->

<property name="viewResolvers">

<list>

<bean class="org.springframework.web.servlet.view.XmlViewResolver">

<property name="location" value="spreadsheet-views.xml"/>

</bean>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="WEB-INF/views"/>

<property name="suffix" value=".jsp"/>

</bean>

</list>

</property>

</bean>

Separate profile :

<!-- Maps to a bean called "accounts/list" in "spreadsheet-views.xml" -->

<bean class="org.springframework.web.servlet.view.XmlViewResolver"> <property name="order" value="1"/>

<property name="location" value="WEB-INF/spring/spreadsheet-views.xml"/>

</bean>

<!-- Maps to "WEB-INF/views/accounts/list.jsp" -->

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="order" value="2"/>

<property name="prefix" value="WEB-INF/views"/>

<property name="suffix" value=".jsp"/>

</bean>

And in WEB-INF/spring/spreadsheet-beans.xml you will find

<bean id="accounts/list" class="rewardsonline.accounts.AccountExcelView"/>

<!-- View resolver that delegates to other view resolvers based on the content type -->

<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver"> <!-- All configuration is now done by the manager - since Spring V3.2 -->

<property name="contentNegotiationManager" ref="cnManager"/>

</bean> <!--

Setup a simple strategy:

1. Only path extension is taken into account, Accept headers are ignored. 2. Return HTML by default when not sure. -->

<bean id="cnManager" class="org.springframework.web.accept.ContentNegotiationManagerFactoryBean"> <property name="ignoreAcceptHeader" value="true"/> <property name="defaultContentType" value="text/html" /> </bean>

The CNVR automatically goes to every other view resolver bean defined to Spring and asks it for a View instance corresponding to the view-name returned by the controller - in this case accounts/list.  Each View ‘knows’ what sort of content it can generate because there is a getContentType() method on it (inherited from the View interface).  The JSP page is rendered by a JstlView (returned by the InternalResourceViewResolver) and its content-type is text/html, whilst the AccountExcelView generates application/vnd.ms-excel.

How the CNVR is actually configured is delegated to the ContentNegotiationManager which is created in turn via the configurer (Java Configuration) or one of Spring’s many factory beans (XML).

The last piece of the puzzle is: how does the CNVR know what content-type was requested? Because the content-negotiation strategy tells it what to do: either a URL suffix is recognized, or a URL parameter or an Accept header. Exactly the same strategy setup described in the previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/), reused by the CNVR.

Note that when content-negotiation strategies were introduced by Spring 3.0 they only applied to selecting Views. Since 3.2 this facility is available across the board (as per my previous [post](http://blog.springsource.org/2013/05/11/content-negotiation-using-spring-mvc/)). The examples in this post use Spring 3.2 and may be different to older examples you have seen before. In particular most of the properties for configuring the content-negotiation strategy are now on the ContentNegotiationManagerFactoryBean and not on the ContentNegotiatingViewResolver. The properties on the CNVR are now deprecated in favor of those on the manager but the CNVR itself works exactly the same way that it always did.

**Configuring the Content Negotiating View Resolver**

By default the CNVR automatically detects all ViewResolvers defined to Spring and negotiates between them. If you prefer, the CNVR itself has a viewResolvers property so you can tell it *explicitly* which view resolvers to use. This makes it obvious that the CNVR is the master resolver and the others are subordinate to it. Note that the order property is no longer needed.

<bean class="org.springframework.web.servlet.view.ContentNegotiatingViewResolver">

<property name="contentNegotiationManager" ref="cnManager"/>

<!-- Define the view resolvers explicitly -->

<property name="viewResolvers">

<list>

<bean class="org.springframework.web.servlet.view.XmlViewResolver">

<property name="location" value="spreadsheet-views.xml"/>

</bean>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

<property name="prefix" value="WEB-INF/views"/>

<property name="suffix" value=".jsp"/>

</bean>

</list>

</property>

</bean>

# View Resolvers:

Below are the important viewresolvers provided by spring framework:

1. **AbstractCachingViewResolver :** Abstract view resolver that caches views. Often views need preparation before they can be used; extending this view resolver provides caching.
2. **XmlViewResolver :**Implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring’s XML bean factories. The default configuration file is /WEB-INF/views.xml.
3. **ResourceBundleViewResolver :**Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name. Typically you define the bundle in a properties file, located in the classpath. The default file name is views.properties.
4. **UrlBasedViewResolver :**Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.
5. **InternalResourceViewResolver :**Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..).
6. **VelocityViewResolver/FreeMarkerViewResolver :**Convenient subclass of UrlBasedViewResolver that supports VelocityView (in effect, Velocity templates) or FreeMarkerView ,respectively, and custom subclasses of them.
7. **ContentNegotiatingViewResolver :**Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept header.

**ResourceBundleViewResolver :**

By default, ResourceBundleViewResolver loads view names from “views.properties” file present into classpath but this location can be overridden through the **basename** property.

<bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">

    <property name="basename" value="views" />

</bean>

In views.properties resource bundle, you can declare view beans in the format of properties

//Equivalent resource bundle properites

home.(class)=org.springframework.web.servlet.view.JstlView

home.url=/WEB-INF/jsp/home.jsp

adminHome.(class)=org.springframework.web.servlet.view.JstlView

adminHome.url=/WEB-INF/jsp/admin/home.jsp

logOffRedirect.(class)=org.springframework.web.servlet.view.RedirectView

logOffRedirect.url=home

#### 16.5.3.1 RedirectView

One way to force a redirect as the result of a controller response is for the controller to create and return an instance of Spring's RedirectView. In this case, DispatcherServlet will not use the normal view resolution mechanism, but rather as it has been given the (redirect) view already, will just ask it to do its work.

The RedirectView simply ends up issuing an HttpServletResponse.sendRedirect() call, which will come back to the client browser as an HTTP redirect. All model attributes are simply exposed as HTTP query parameters. This does mean that the model must contain only objects (generally Strings or convertible to Strings) which can be readily converted to a string-form HTTP query parameter.

If using RedirectView and the view is created by the controller itself, it is preferable for the redirect URL to be injected into the controller so that it is not baked into the controller but configured in the context along with the view names.

#### 16.5.3.2 The redirect: prefix

While the use of RedirectView works fine, if the controller itself is creating the RedirectView, there is no getting around the fact that the controller is aware that a redirection is happening. This is really suboptimal and couples things too tightly. The controller should not really care about how the response gets handled... it should generally think only in terms of view names that have been injected into it.

The special redirect: prefix allows this to be achieved. If a view name is returned which has the prefix redirect:, then UrlBasedViewResolver (and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

The net effect is the same as if the controller had returned a RedirectView, but now the controller itself can deal just in terms of logical view names. A logical view name such as redirect:/my/response/controller.html will redirect relative to the current servlet context, while a name such asredirect:http://myhost.com/some/arbitrary/path.html will redirect to an absolute URL. The important thing is that as long as this redirect view name is injected into the controller like any other logical view name, the controller is not even aware that redirection is happening.

#### 16.5.3.3 The forward: prefix

It is also possible to use a special forward: prefix for view names that will ultimately be resolved by UrlBasedViewResolver and subclasses. All this does is create an InternalResourceView (which ultimately does a RequestDispatcher.forward()) around the rest of the view name, which is considered a URL. Therefore, there is never any use in using this prefix when using InternalResourceViewResolver / InternalResourceView anyway (for JSPs for example), but it's of potential use when you are primarily using another view technology, but still want to force a forward to happen to a resource to be handled by the Servlet/JSP engine. (Note that you may also chain multiple view resolvers, instead.)

As with the redirect: prefix, if the view name with the prefix is just injected into the controller, the controller does not have to be aware that anything special is happening in terms of handling the response.