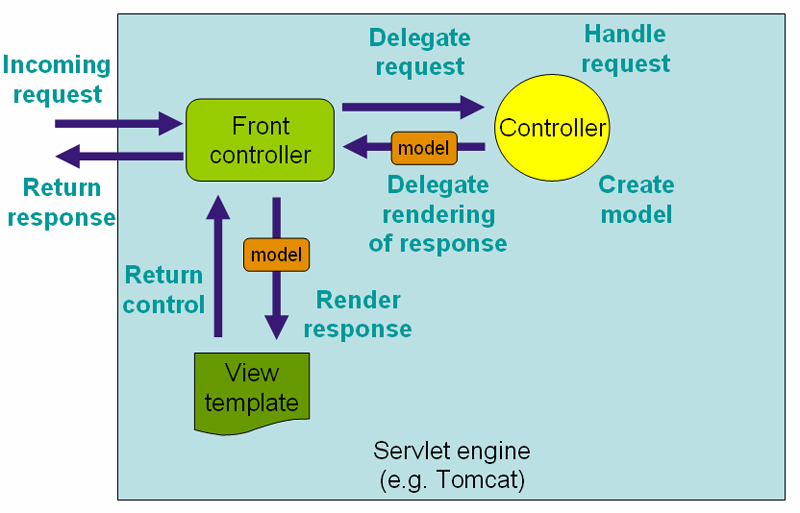
**Intro**

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as support for uploading files. The default handler is based on the @Controller and @RequestMapping annotations, offering a wide range of flexible handling methods

**DispatcherServlet**

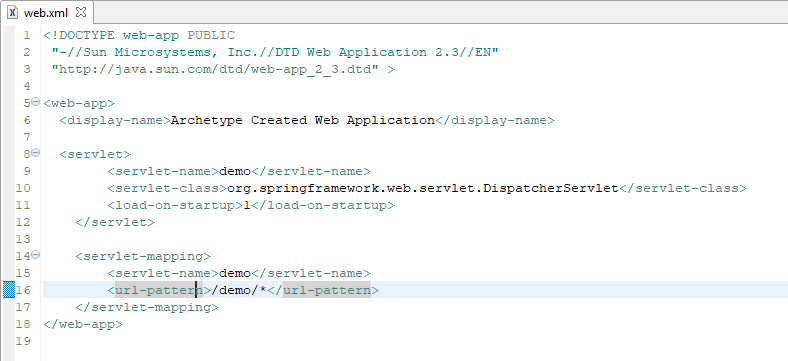
Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central Servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications. Spring's DispatcherServlet however, does more than just that. It is completely integrated with the Spring IoC container and as such allows you to use every other feature that Spring has.

The request processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the following diagram. The pattern-savvy reader will recognize that theDispatcherServlet is an expression of the “Front Controller” design pattern (this is a pattern that Spring Web MVC shares with many other leading web frameworks).

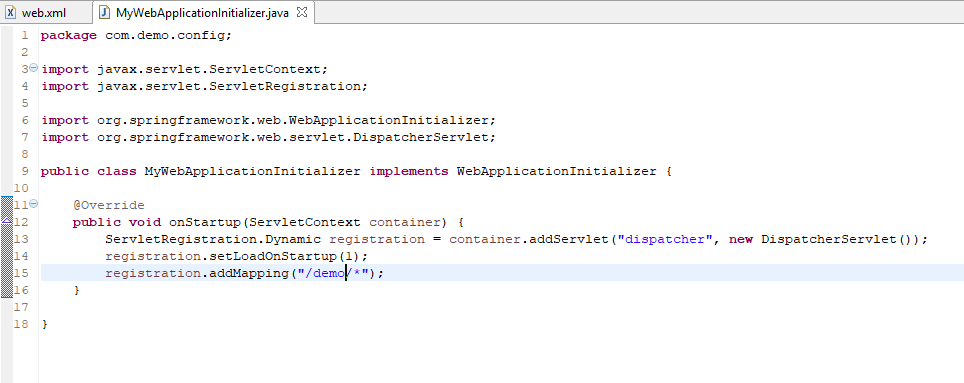


The request processing workflow in Spring Web MVC (high level)

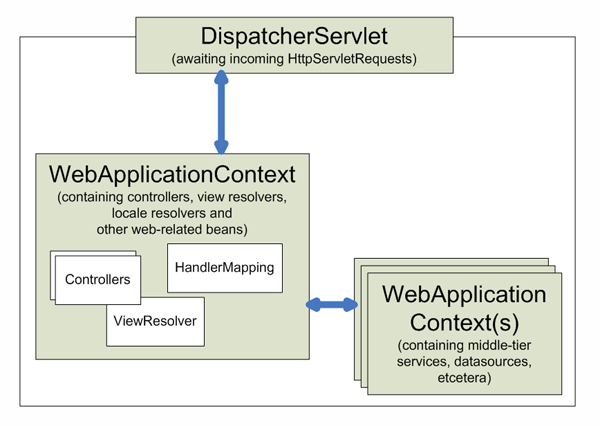
The DispatcherServlet is an actual Servlet (it inherits from the HttpServlet base class), and as such is declared in the web.xml of your web application. You need to map requests that you want the DispatcherServlet to handle, by using a URL mapping in the same web.xml file. This is standard Java EE Servlet configuration; the following example shows such a DispatcherServlet declaration and mapping:



In the preceding example, all requests starting with /demo will be handled by the DispatcherServlet instance named example. In a Servlet 3.0+ environment, you also have the option of configuring the Servlet container programmatically. Below is the code based equivalent of the above web.xml example:



 ApplicationContext instances in Spring can be scoped. In the Web MVC framework, each DispatcherServlet has its own WebApplicationContext, which inherits all the beans already defined in the root WebApplicationContext. These inherited beans can be overridden in the servlet-specific scope, and you can define new scope-specific beans local to a given Servlet instance.

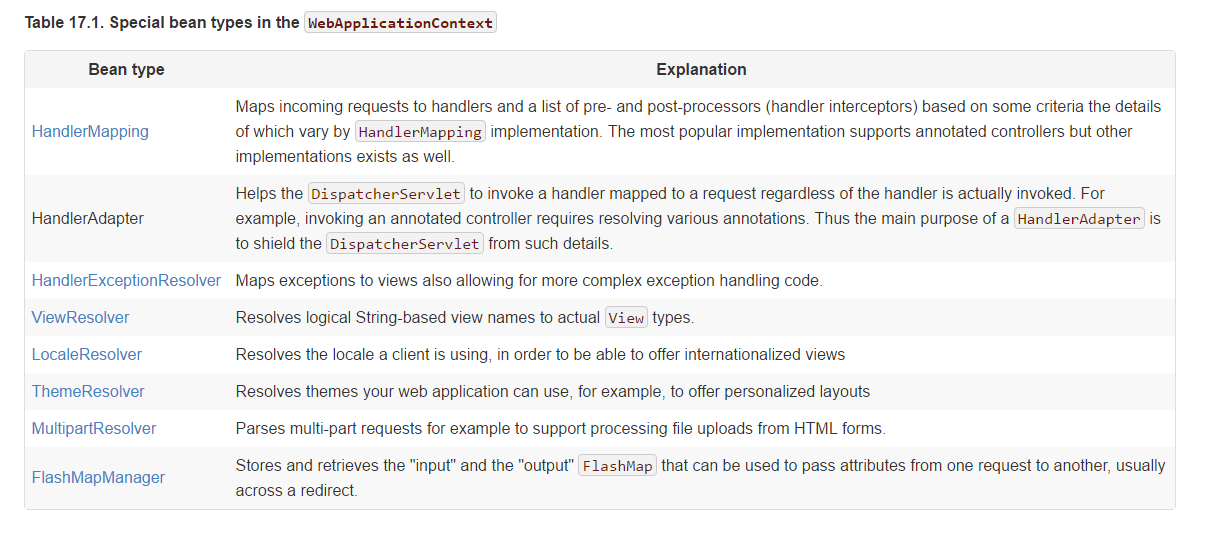


Context hierarchy in Spring Web MVC

Upon initialization of a DispatcherServlet, Spring MVC looks for a file named *[servlet-name]-servlet.xml* in the WEB-INF directory of your web application and creates the beans defined there, overriding the definitions of any beans defined with the same name in the global scope.

With the above Servlet configuration in place, you will need to have a file called /WEB-INF/demo-servlet.xml in your application

**Default Beans in WebApplicationContext**



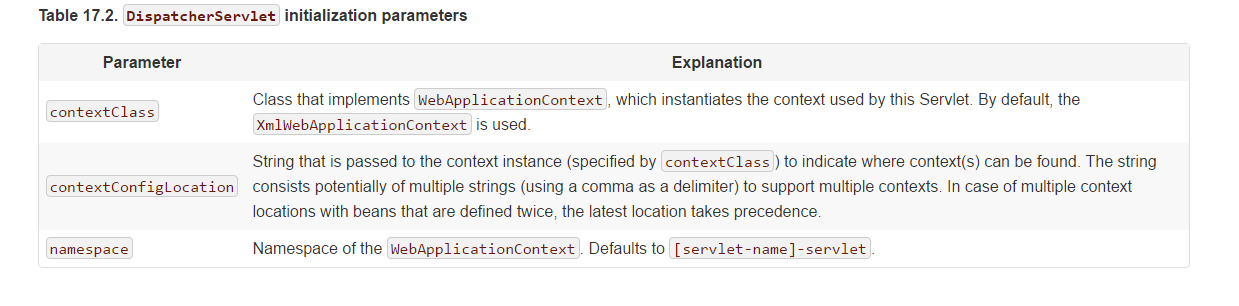
**DispatcherServlet Processing Sequence**

After you set up a DispatcherServlet, and a request comes in for that specific DispatcherServlet, the DispatcherServlet starts processing the request as follows:

* The WebApplicationContext is searched for and bound in the request as an attribute that the controller and other elements in the process can use. It is bound by default under the key DispatcherServlet.WEB\_APPLICATION\_CONTEXT\_ATTRIBUTE.
* The locale resolver is bound to the request to enable elements in the process to resolve the locale to use when processing the request (rendering the view, preparing data, and so on). If you do not need locale resolving, you do not need it.
* The theme resolver is bound to the request to let elements such as views determine which theme to use. If you do not use themes, you can ignore it.
* If you specify a multipart file resolver, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a MultipartHttpServletRequest for further processing by other elements in the process. See Section 17.10, “Spring's multipart (file upload) support” for further information about multipart handling.
* An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) is executed in order to prepare a model or rendering.
* If a model is returned, the view is rendered. If no model is returned, (may be due to a preprocessor or postprocessor intercepting the request, perhaps for security reasons), no view is rendered, because the request could already have been fulfilled.

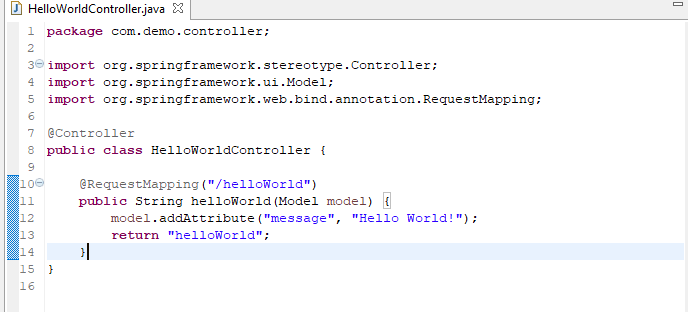
Handler exception resolvers that are declared in the WebApplicationContext pick up exceptions that are thrown during processing of the request. Using these exception resolvers allows you to define custom behaviors to address exceptions.

You can customize individual DispatcherServlet instances by adding Servlet initialization parameters (init-param elements) to the Servlet declaration in the web.xml file. See the following table for the list of supported parameters.



**Controllers**

Controllers provide access to the application behavior that you typically define through a service interface. Controllers interpret user input and transform it into a model that is represented to the user by the view. Spring implements a controller in a very abstract way, which enables you to create a wide variety of controllers.



The @Controller annotation indicates that a particular class serves the role of a controller. Spring does not require you to extend any controller base class or reference the Servlet API. However, you can still reference Servlet-specific features if you need to.

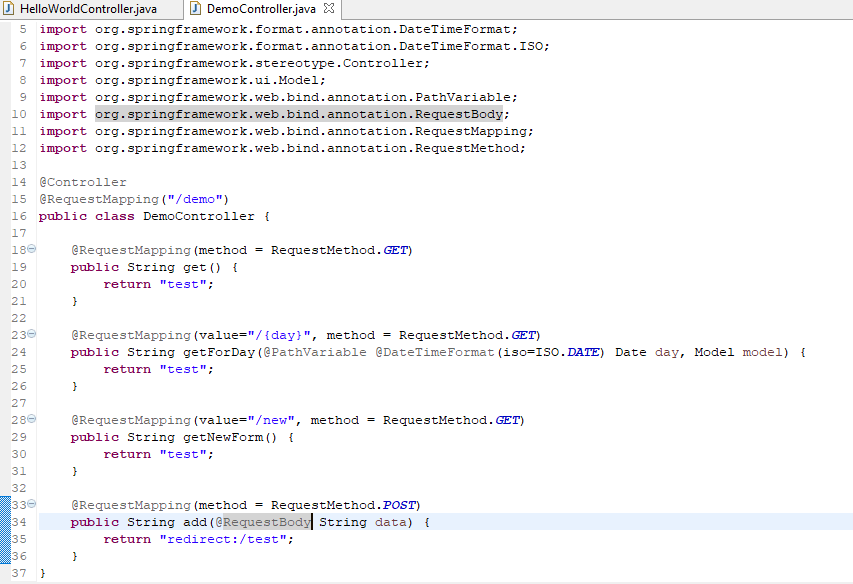
The @Controller annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotations (see the next section).

You can define annotated controller beans explicitly, using a standard Spring bean definition in the dispatcher's context. However, the @Controller stereotype also allows for autodetection, aligned with Spring general support for detecting component classes in the classpath and auto-registering bean definitions for them.

To enable autodetection of such annotated controllers, you add component scanning to your configuration

**@RequestMapping**

you use the @RequestMapping annotation to map URLs such as /appointments onto an entire class or a particular handler method. Typically the class-level annotation maps a specific request path (or path pattern) onto a form controller, with additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition.



Spring 3.1 introduced a new set of support classes for @RequestMapping methods called RequestMappingHandlerMapping and RequestMappingHandlerAdapter respectively. They are recommended for use and even required to take advantage of new features in Spring MVC 3.1 and going forward. The new support classes are enabled by default by the MVC namespace and the MVC Java config but must be configured explicitly if using neither. This section describes a few important differences between the old and the new support classes.

Prior to Spring 3.1, type and method-level request mappings were examined in two separate stages -- a controller was selected first by the DefaultAnnotationHandlerMapping and the actual method to invoke was narrowed down second by the AnnotationMethodHandlerAdapter.

With the new support classes in Spring 3.1, the RequestMappingHandlerMapping is the only place where a decision is made about which method should process the request. Think of controller methods as a collection of unique endpoints with mappings for each method derived from type and method-level @RequestMapping information.

This enables some new possibilities. For once a HandlerInterceptor or a HandlerExceptionResolver can now expect the Object-based handler to be a HandlerMethod, which allows them to examine the exact method, its parameters and associated annotations. The processing for a URL no longer needs to be split across different controllers.

**URI Template**

URI templates can be used for convenient access to selected parts of a URL in a @RequestMapping method.

A URI Template is a URI-like string, containing one or more variable names. When you substitute values for these variables, the template becomes a URI. The [proposed RFC](http://bitworking.org/projects/URI-Templates/) for URI Templates defines how a URI is parameterized. For example, the URI Template http://www.example.com/users/{userId} contains the variable userId. Assigning the value fred to the variable yields http://www.example.com/users/fred.

In Spring MVC you can use the @PathVariable annotation on a method argument to bind it to the value of a URI template variable:

*@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)*

**public** String findOwner(**@PathVariable** String ownerId, Model model) {

Owner owner = ownerService.findOwner(ownerId);

model.addAttribute("owner", owner);

**return** "displayOwner";

}

The URI Template "/owners/{ownerId}" specifies the variable name ownerId. When the controller handles this request, the value of ownerId is set to the value found in the appropriate part of the URI. For example, when a request comes in for /owners/fred, the value of ownerId is fred.

|  |
| --- |
|  |
| To process the @PathVariable annotation, Spring MVC needs to find the matching URI template variable by name. You can specify it in the annotation:  *@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)*  **public** String findOwner(**@PathVariable**("ownerId") String theOwner, Model model) {  *// implementation omitted*  }  Or if the URI template variable name matches the method argument name you can omit that detail. As long as your code is not compiled without debugging information, Spring MVC will match the method argument name to the URI template variable name:  *@RequestMapping(value="/owners/{ownerId}", method=RequestMethod.GET)*  **public** String findOwner(**@PathVariable** String ownerId, Model model) {  *// implementation omitted*  } |

A method can have any number of @PathVariable annotations:

*@RequestMapping(value="/owners/{ownerId}/pets/{petId}", method=RequestMethod.GET)*

**public** String findPet(**@PathVariable** String ownerId, **@PathVariable** String petId, Model model) {

Owner owner = ownerService.findOwner(ownerId);

Pet pet = owner.getPet(petId);

model.addAttribute("pet", pet);

**return** "displayPet";

}

When a @PathVariable annotation is used on a Map<String, String> argument, the map is populated with all URI template variables.

A URI template can be assembled from type and path level *@RequestMapping* annotations. As a result the findPet() method can be invoked with a URL such as /owners/42/pets/21.

*@Controller*

@RequestMapping(**"/owners/{ownerId}"**)

**public** **class** RelativePathUriTemplateController {

@RequestMapping(**"/pets/{petId}"**)

**public** **void** findPet(*@PathVariable* String ownerId, *@PathVariable* String petId, Model model) {

*// implementation omitted*

}

}

A @PathVariable argument can be of **any simple type** such as int, long, Date, etc. Spring automatically converts to the appropriate type or throws a TypeMismatchException if it fails to do so. You can also register support for parsing additional data types. See [the section called “Method Parameters And Type Conversion”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-ann-typeconversion) and [the section called “Customizing WebDataBinder initialization”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-ann-webdatabinder).

**URI Template Patterns with Regular Expressions**

Sometimes you need more precision in defining URI template variables. Consider the URL "/spring-web/spring-web-3.0.5.jar". How do you break it down into multiple parts?

The @RequestMapping annotation supports the use of regular expressions in URI template variables. The syntax is {varName:regex} where the first part defines the variable name and the second - the regular expression.For example:

*@RequestMapping("/spring-web/{symbolicName:[a-z-]+}-{version:\\d\\.\\d\\.\\d}{extension:\\.[a-z]+}")*

**public** **void** handle(*@PathVariable* String version, *@PathVariable* String extension) {

*// ...*

}

}

**Path Patterns**

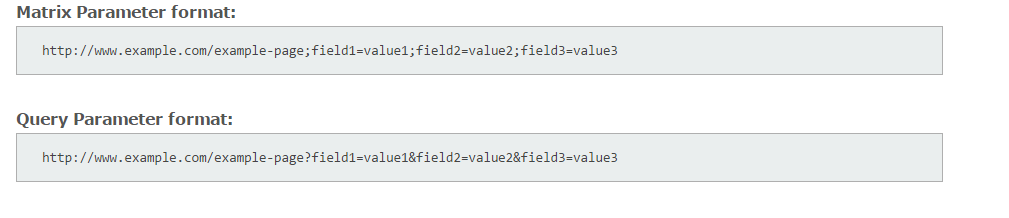
In addition to URI templates, the @RequestMapping annotation also supports Ant-style path patterns (for example, /myPath/\*.do). A combination of URI templates and Ant-style globs is also supported (for example, /owners/\*/pets/{petId}).

**Patterns with Placeholders**

Patterns in @RequestMapping annotations support ${...} placeholders against local properties and/or system properties and environment variables. This may be useful in cases where the path a controller is mapped to may need to be customized through configuration

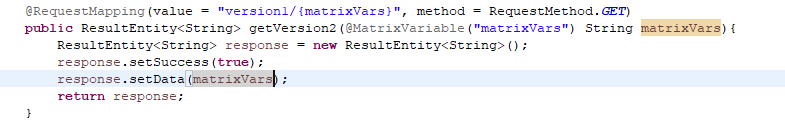
**Matrix Variables**

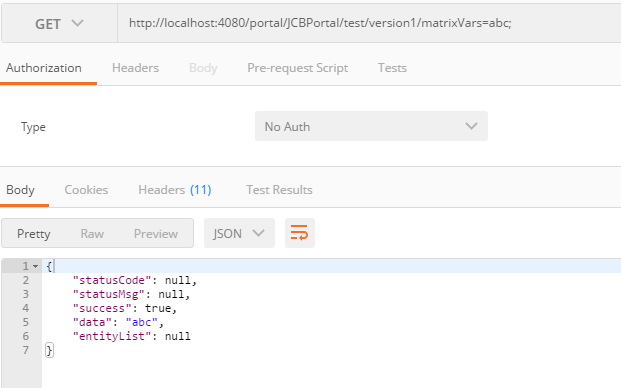
Matrix parameters are alternative to Query parameters. Both can insert optional parameters in a URL. The @MatrixVariable annotation is used to bind the matrix variables to method parameters in a controller.



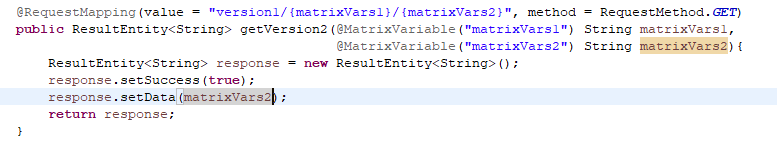
The following examples demonstrate how the @MatrixVariable annotation works in Spring application.

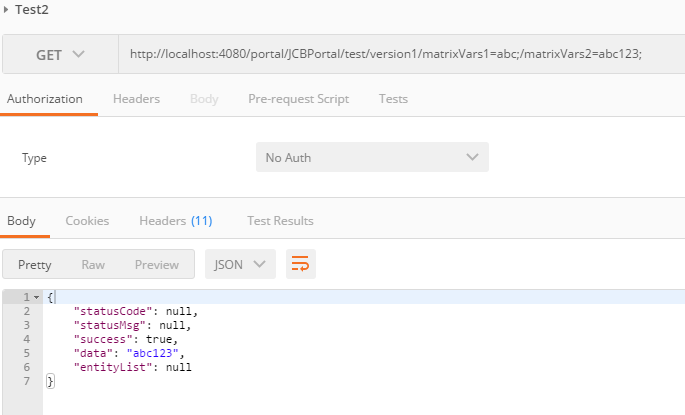
1)



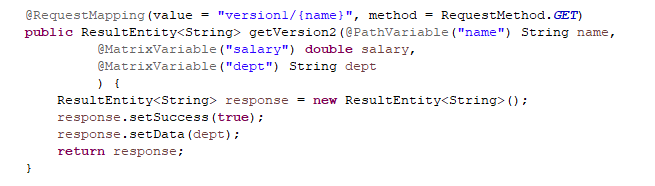


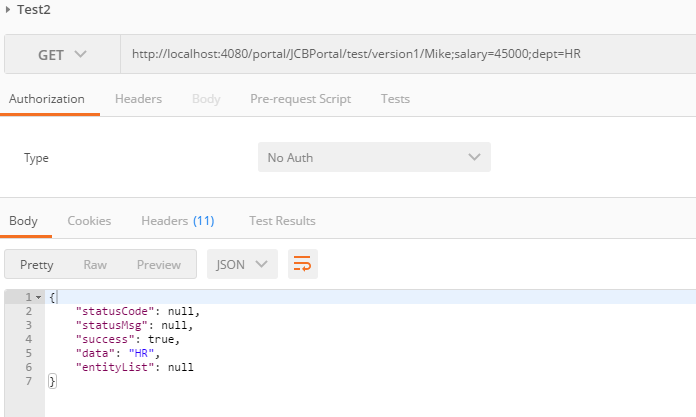
2)



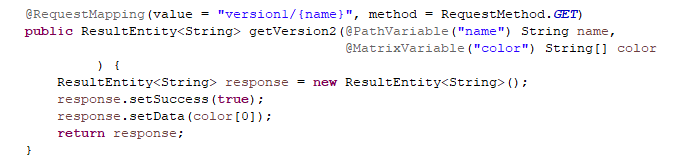


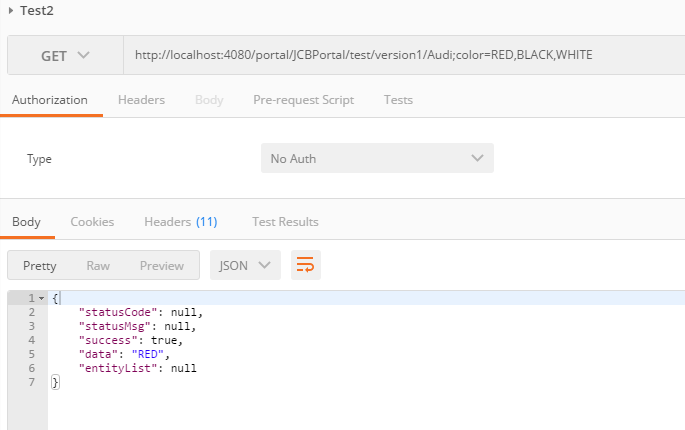
3)



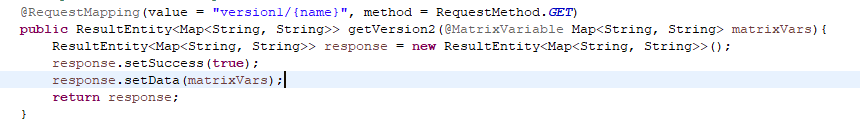


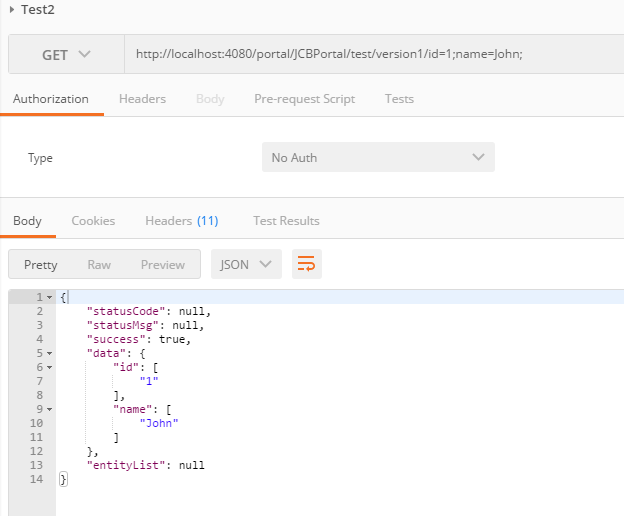
4)



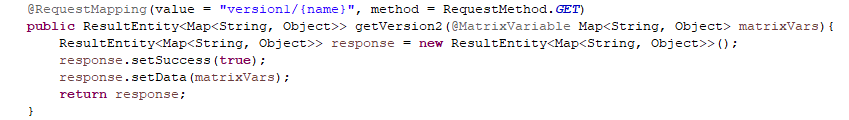


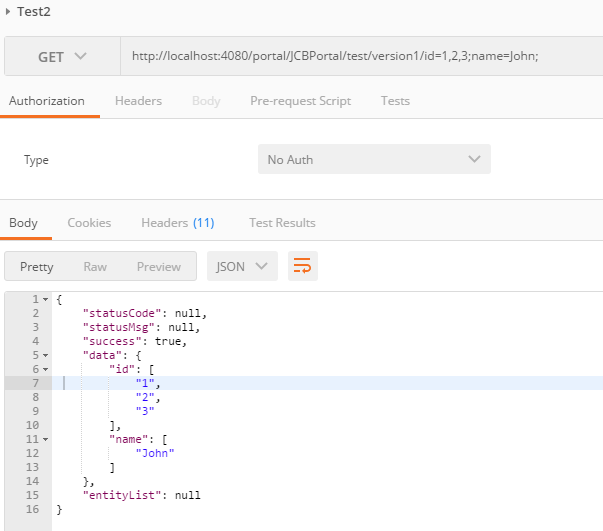
5)



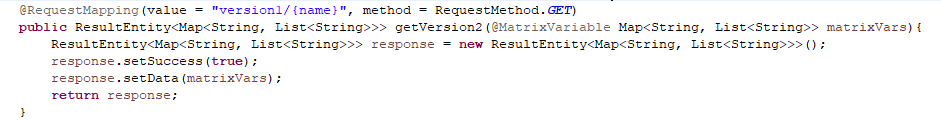


6)





7)





#### Consumable Media Types

You can narrow the primary mapping by specifying a list of consumable media types. The request will be matched only if the Content-Type request header matches the specified media type. For example:

*@Controller*

@RequestMapping(value = "/pets", method = RequestMethod.POST, **consumes="application/json"**)

**public** **void** addPet(*@RequestBody* Pet pet, Model model) {

*// implementation omitted*

}

Consumable media type expressions can also be negated as in !text/plain to match to all requests other than those with Content-Type of text/plain.

#### Producible Media Types

You can narrow the primary mapping by specifying a list of producible media types. The request will be matched only if the Accept request header matches one of these values. Furthermore, use of the produces condition ensures the actual content type used to generate the response respects the media types specified in the produces condition. For example:

*@Controller*

@RequestMapping(value = "/pets/{petId}", method = RequestMethod.GET, **produces="application/json"**)

*@ResponseBody*

**public** Pet getPet(*@PathVariable* String petId, Model model) {

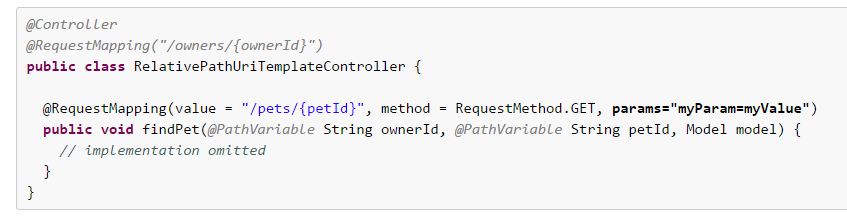
*// implementation omitted*

}

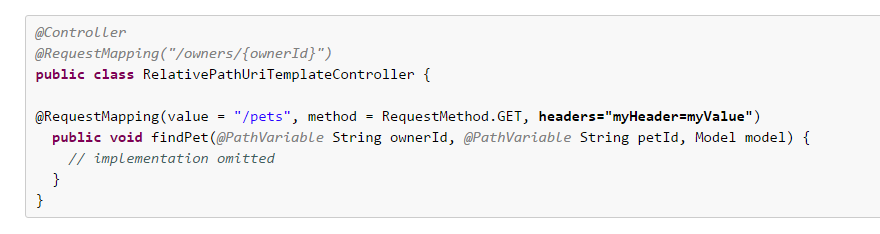
Just like with consumes, producible media type expressions can be negated as in !text/plain to match to all requests other than those with an Accept header value oftext/plain.

**Request Parameters and Header Values**

You can narrow request matching through request parameter conditions such as "myParam", "!myParam", or "myParam=myValue". The first two test for request parameter presence/absence and the third for a specific parameter value. Here is an example with a request parameter value condition:



The same can be done to test for request header presence/absence or to match based on a specific request header value:



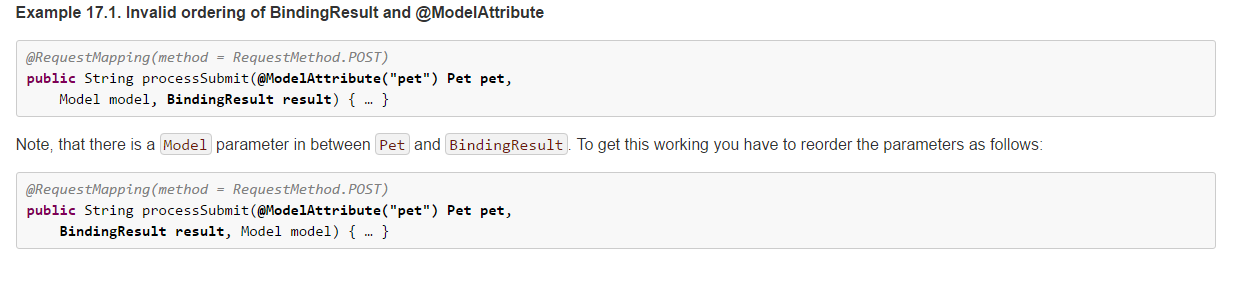
**Supported method argument types**

* Request or response objects (Servlet API). Choose any specific request or response type, for example ServletRequest or HttpServletRequest.
* Session object (Servlet API): of type HttpSession. An argument of this type enforces the presence of a corresponding session. As a consequence, such an argument is never null.

|  |
| --- |
|  |
| Session access may not be thread-safe, in particular in a Servlet environment. Consider setting the RequestMappingHandlerAdapter's "synchronizeOnSession" flag to "true" if multiple requests are allowed to access a session concurrently. |

* org.springframework.web.context.request.WebRequest or org.springframework.web.context.request.NativeWebRequest. Allows for generic request parameter access as well as request/session attribute access, without ties to the native Servlet/Portlet API.
* java.util.Locale for the current request locale, determined by the most specific locale resolver available, in effect, the configured LocaleResolver in a Servlet environment.
* java.io.InputStream / java.io.Reader for access to the request's content. This value is the raw InputStream/Reader as exposed by the Servlet API.
* java.io.OutputStream / java.io.Writer for generating the response's content. This value is the raw OutputStream/Writer as exposed by the Servlet API.
* java.security.Principal containing the currently authenticated user.
* @PathVariable annotated parameters for access to URI template variables. See [the section called “URI Template Patterns”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-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”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-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”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-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 usingHttpMessageConverters. See [the section called “Mapping the request body with the @RequestBody annotation”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-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”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-multipart-forms-non-browsers) and [Section 17.10, “Spring's multipart (file upload) support”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-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 usingHttpMessageConverters. See [the section called “Using HttpEntity<?>”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-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.
* org.springframework.web.servlet.mvc.support.RedirectAttributes to specify the exact set of attributes to use in case of a redirect and also to add flash attributes (attributes stored temporarily on the server-side to make them available to the request after the redirect). RedirectAttributes is used instead of the implicit model if the method returns a "redirect:" prefixed view name or RedirectView.
* Command or form objects to bind request parameters to bean properties (via setters) or directly to fields, with customizable type conversion, depending on@InitBinder methods and/or the HandlerAdapter configuration. See the webBindingInitializer property on RequestMappingHandlerAdapter. Such command objects along with their validation results will be exposed as model attributes by default, using the command class class name - e.g. model attribute "orderAddress" for a command object of type "some.package.OrderAddress". The ModelAttribute annotation can be used on a method argument to customize the model attribute name used.
* org.springframework.validation.Errors / org.springframework.validation.BindingResult validation results for a preceding command or form object (the immediately preceding method argument).
* org.springframework.web.bind.support.SessionStatus status handle for marking form processing as complete, which triggers the cleanup of session attributes that have been indicated by the @SessionAttributes annotation at the handler type level.
* org.springframework.web.util.UriComponentsBuilder a builder for preparing a URL relative to the current request's host, port, scheme, context path, and the literal part of the servlet mapping.

The Errors or BindingResult parameters have to follow the model object that is being bound immediately as the method signature might have more that one model object and Spring will create a separate BindingResult instance for each of them so the following sample won't work:

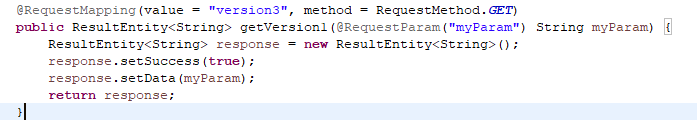


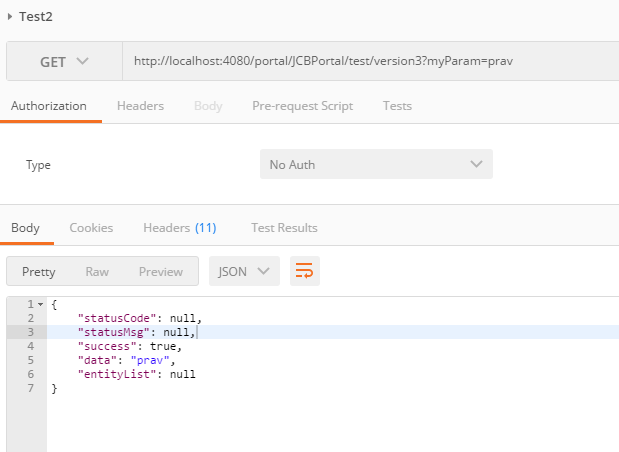
**Supported method return types**

* A ModelAndView object, with the model implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.
* A Model object, with the view name implicitly determined through a RequestToViewNameTranslator and the model implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.
* A Map object for exposing a model, with the view name implicitly determined through a RequestToViewNameTranslator and the model implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.
* A View object, with the model implicitly determined through command objects and @ModelAttribute annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).
* A String value that is interpreted as the logical view name, with the model implicitly determined through command objects and @ModelAttribute annotated reference data accessor methods. The handler method may also programmatically enrich the model by declaring a Model argument (see above).
* void if the method handles the response itself (by writing the response content directly, declaring an argument of type ServletResponse / HttpServletResponse for that purpose) or if the view name is supposed to be implicitly determined through a RequestToViewNameTranslator (not declaring a response argument in the handler method signature).
* If the method is annotated with @ResponseBody, the return type is written to the response HTTP body. The return value will be converted to the declared method argument type using HttpMessageConverters. See [the section called “Mapping the response body with the @ResponseBody annotation”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-ann-responsebody).
* A HttpEntity<?> or ResponseEntity<?> object to provide access to the Servlet response HTTP headers and contents. The entity body will be converted to the response stream using HttpMessageConverters. See [the section called “Using HttpEntity<?>”](https://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html#mvc-ann-httpentity).
* A Callable<?> can be returned when the application wants to produce the return value asynchronously in a thread managed by Spring MVC.
* A DeferredResult<?> can be returned when the application wants to produce the return value from a thread of its own choosing.
* Any other return type is considered to be a single model attribute to be exposed to the view, using the attribute name specified through @ModelAttribute at the method level (or the default attribute name based on the return type class name). The model is implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods.

**@RequestParam**

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

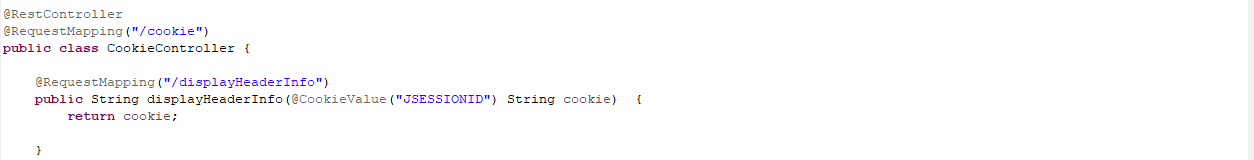




Parameters using this annotation are required by default, but you can specify that a parameter is optional by setting @RequestParam's required attribute to false (e.g., @RequestParam(value="id", required=false)).

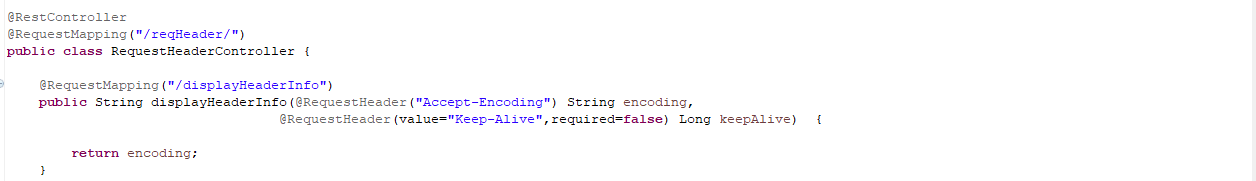
**@CookieValue**

The @CookieValue annotation allows a method parameter to be bound to the value of an HTTP cookie.



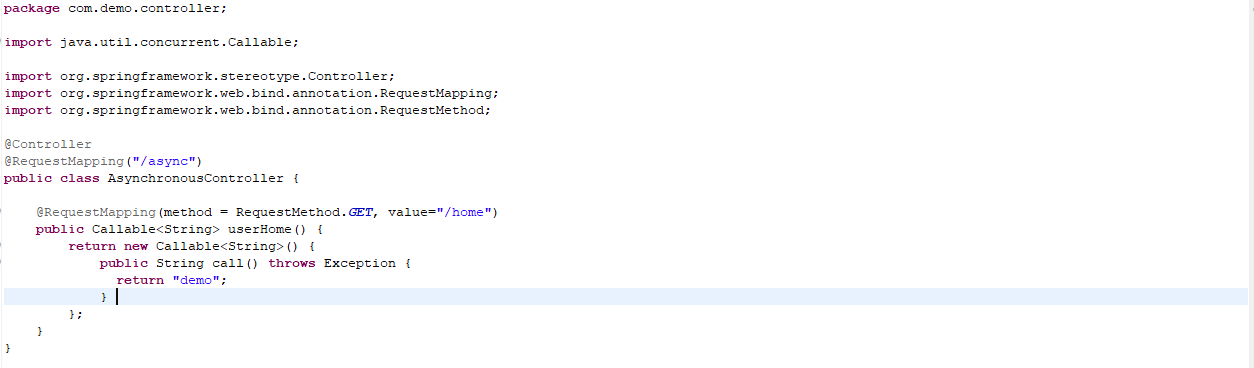
**@RequestHeader**

The @RequestHeader annotation allows a method parameter to be bound to a request header.



**Asynchronous Request Processing**

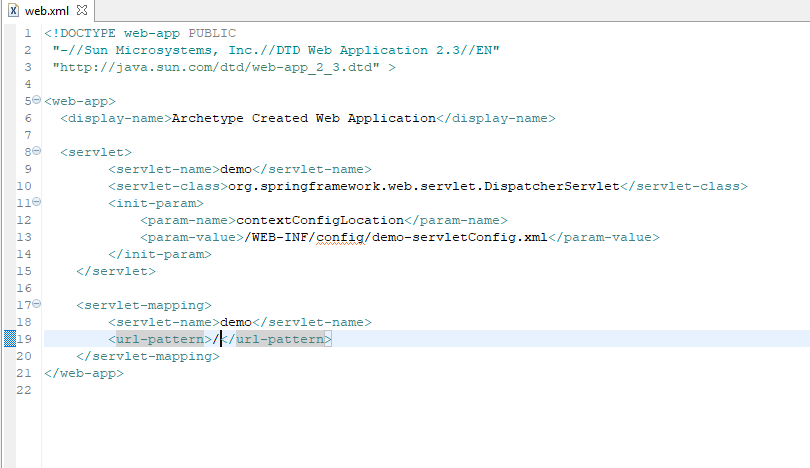
Spring MVC 3.2 introduced Servlet 3 based asynchronous request processing. Instead of returning a value, as usual, a controller method can now return a java.util.concurrent.Callable and produce the return value from a separate thread. Meanwhile the main Servlet container thread is released and allowed to process other requests. Spring MVC invokes the Callable in a separate thread with the help of a TaskExecutor and when the Callable returns, the request is dispatched back to the Servlet container to resume processing with the value returned by the Callable. Here is an example controller method:



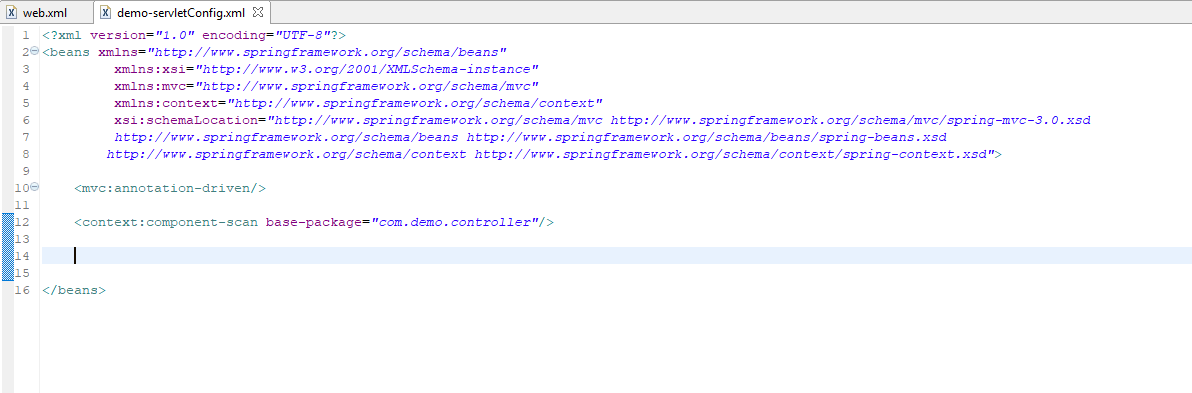
If you get error enable Async, make sure filter is also async or disable all filters



**Configuring web.xml**



**Configuring config file**

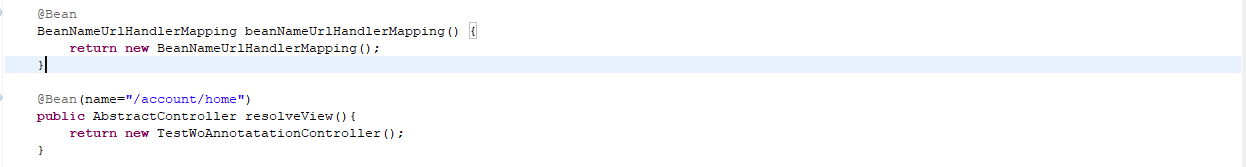


**Handler Mapping**

**BeanNameUrlHandlerMapping**

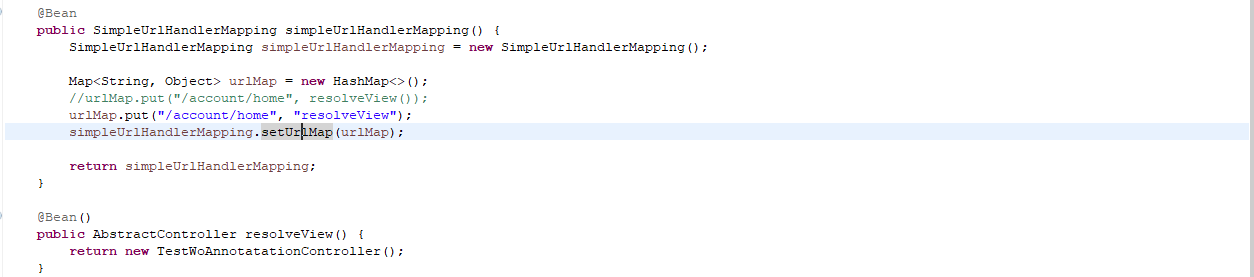
BeanNameUrlHandlerMapping is the default HandlerMapping implementation. BeanNameUrlHandlerMapping maps request URLs to beans with the same name.

This particular mapping has support for direct name matching and also for pattern matching using the “\*” pattern.

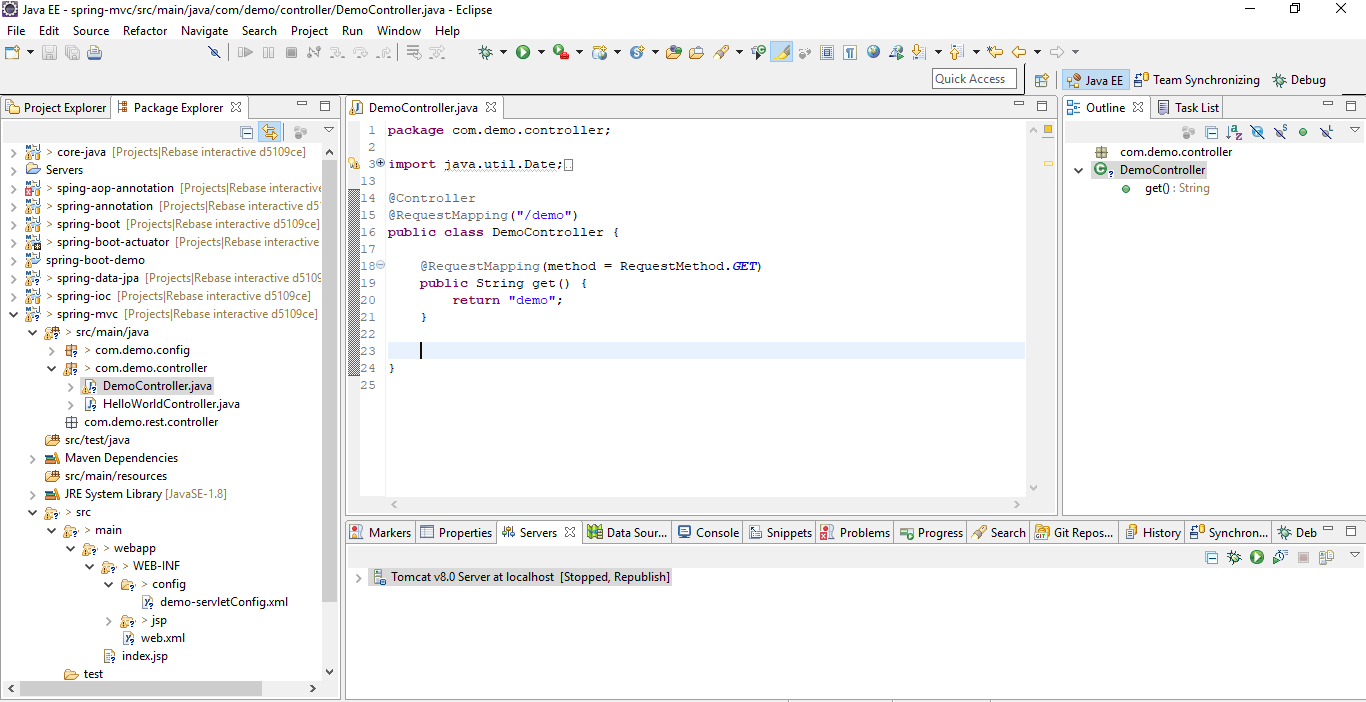


**SimpleUrlHandlerMapping**

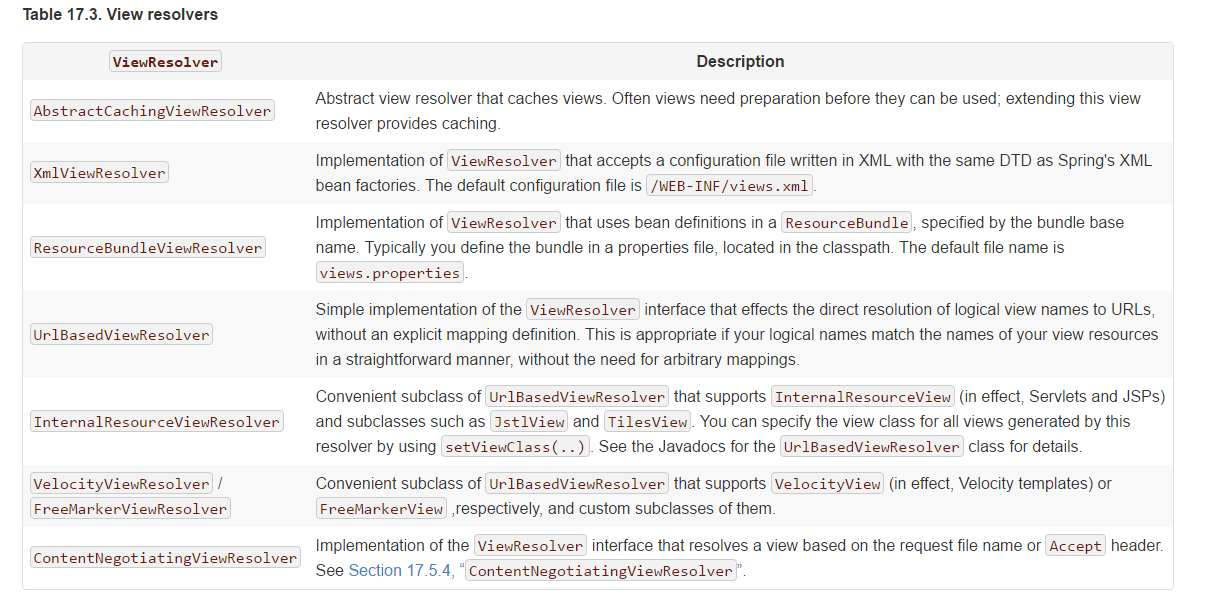
SimpleUrlHandlerMapping is the most flexible HandlerMapping implementation. It allows for direct and declarative mapping between either bean instances and URLs or between bean names and URLs.



**Configuring Controllers**

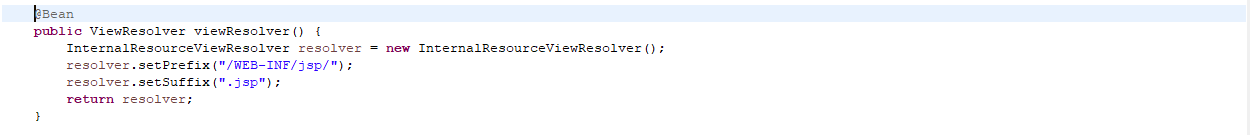


**Creating View Resolvers and Views**



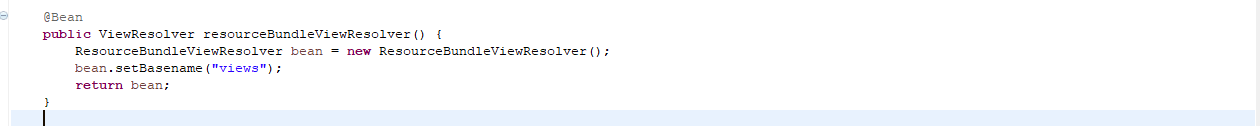
**InternalResourceViewResolver**

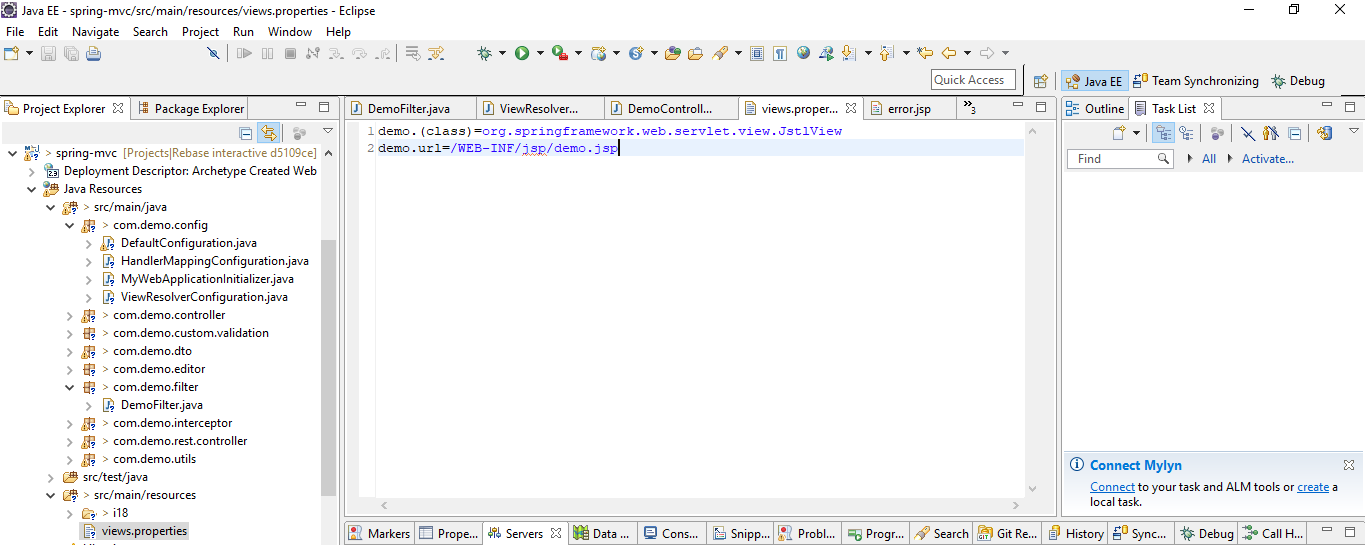
The InternalResourceViewResolver maps the jsp and html files in the WebContent/WEB-INF/ folder. It allows us to set properties such as prefix or suffix to the view name to generate the final view page URL. It is configured as shown below in mvc-dispatcher-servlet.xml.

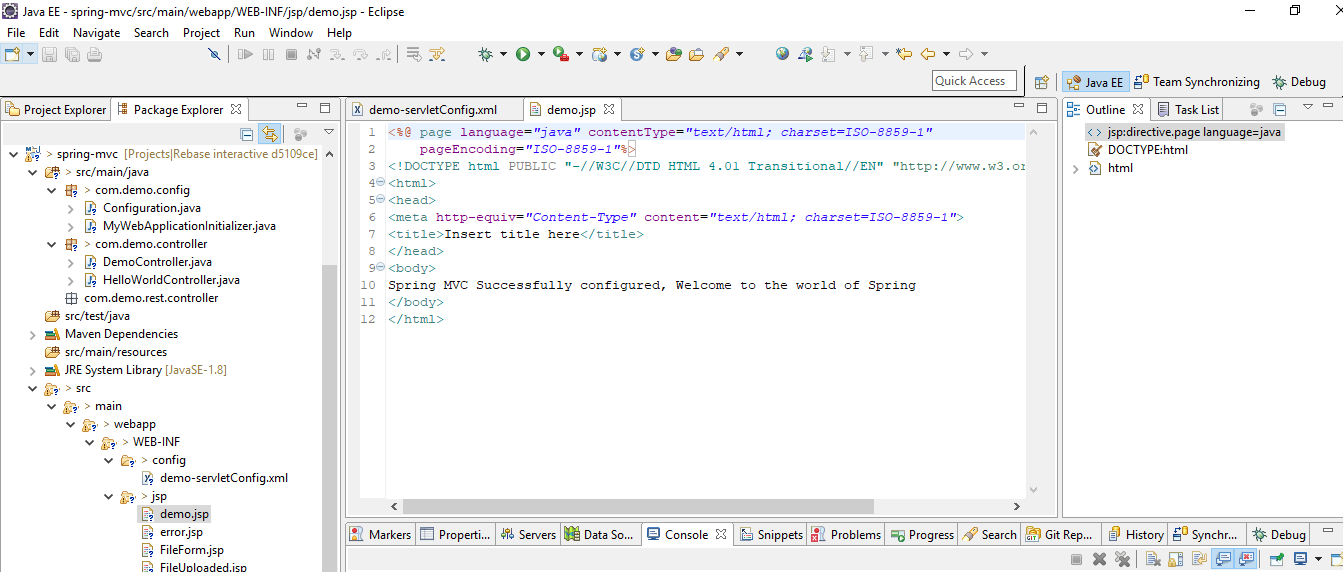


**ResourceBundleViewResolver**

The ResourceBundleViewResolver uses bean definitions in a ResourceBundle, that is specified by the bundle basename. The bundle is typically defined in a properties file, located in the classpath. Below is the views.properties file.

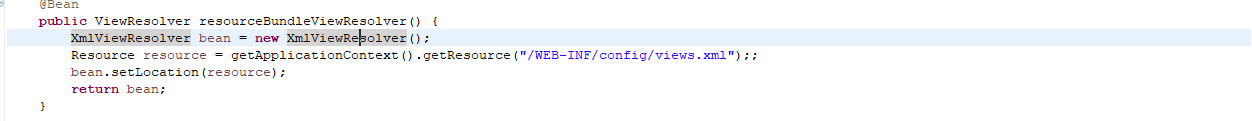


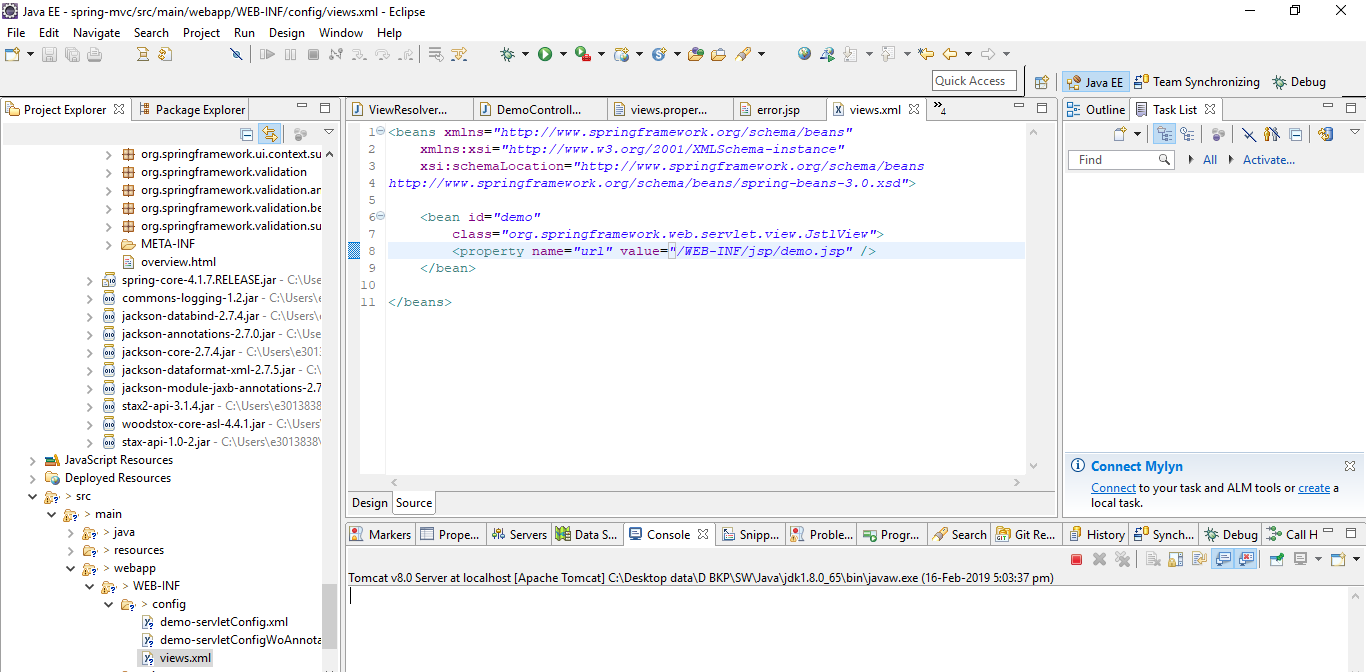




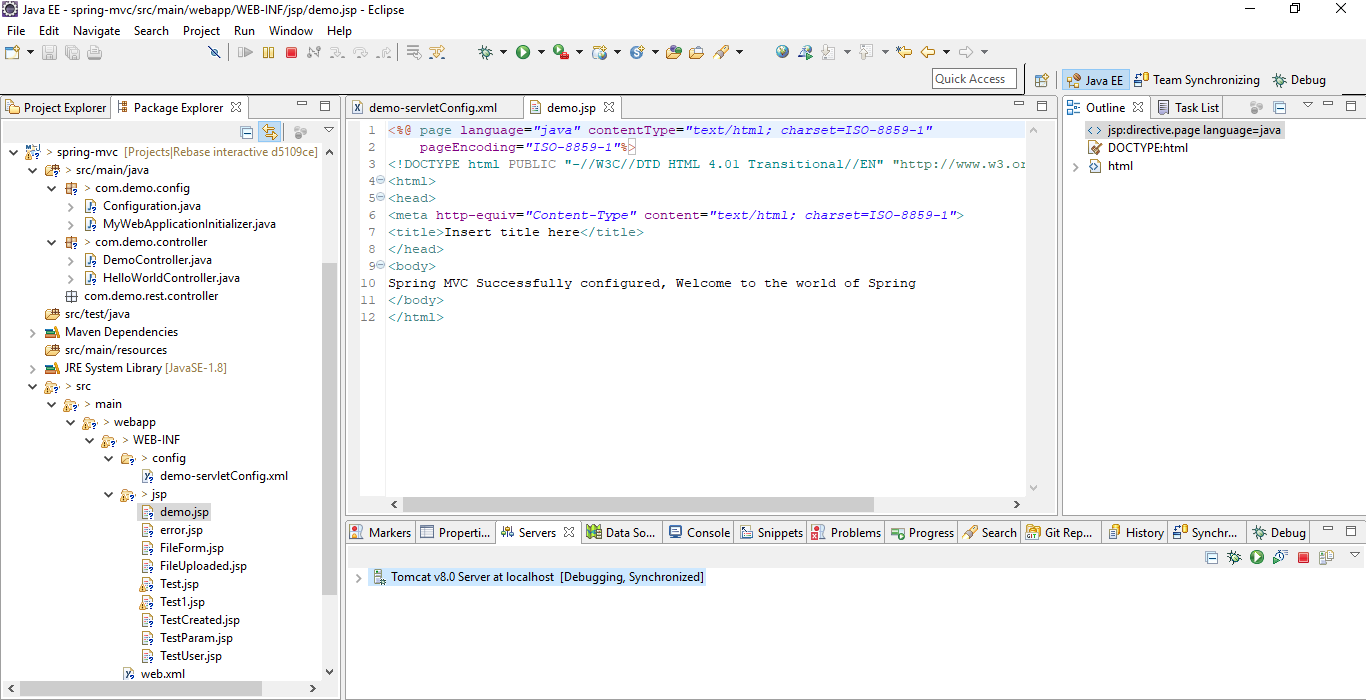
**XmlViewResolver**

XmlViewResolver is an implementation of ViewResolver that accepts a configuration file written in XML, where the view implementation and the url of the jsp file are set. Below is the configuration file, views.xml.



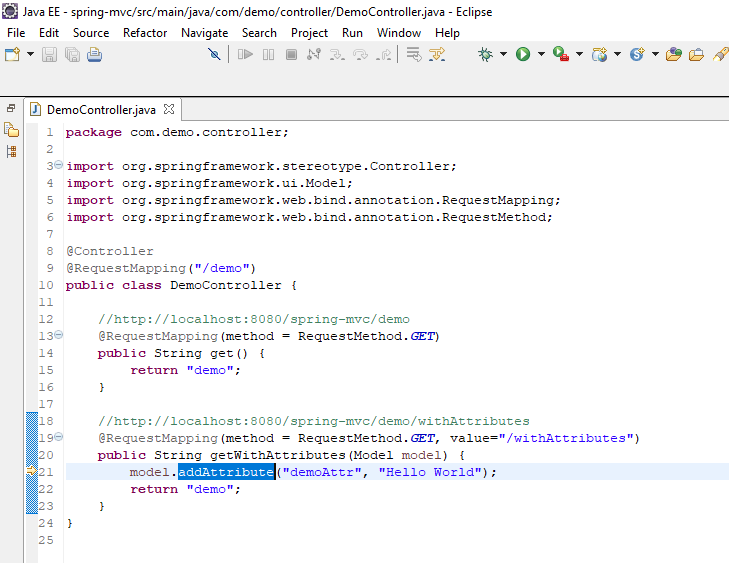


Now run and see



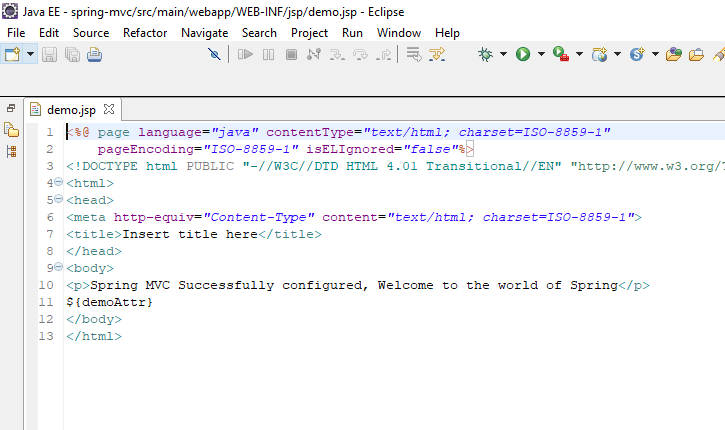


**Adding attributes in Spring MVC**

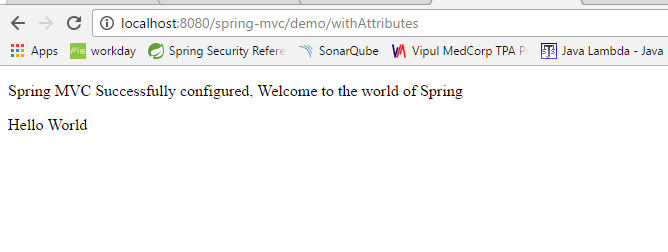


**JSP**

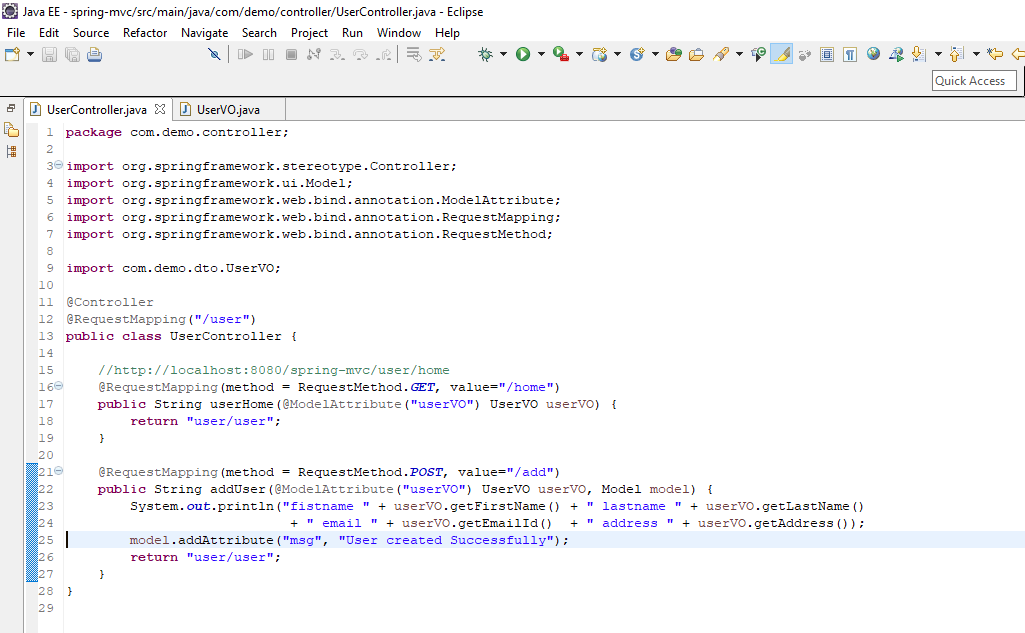
If not working, add isELIgnored="false" in <%@Page %>

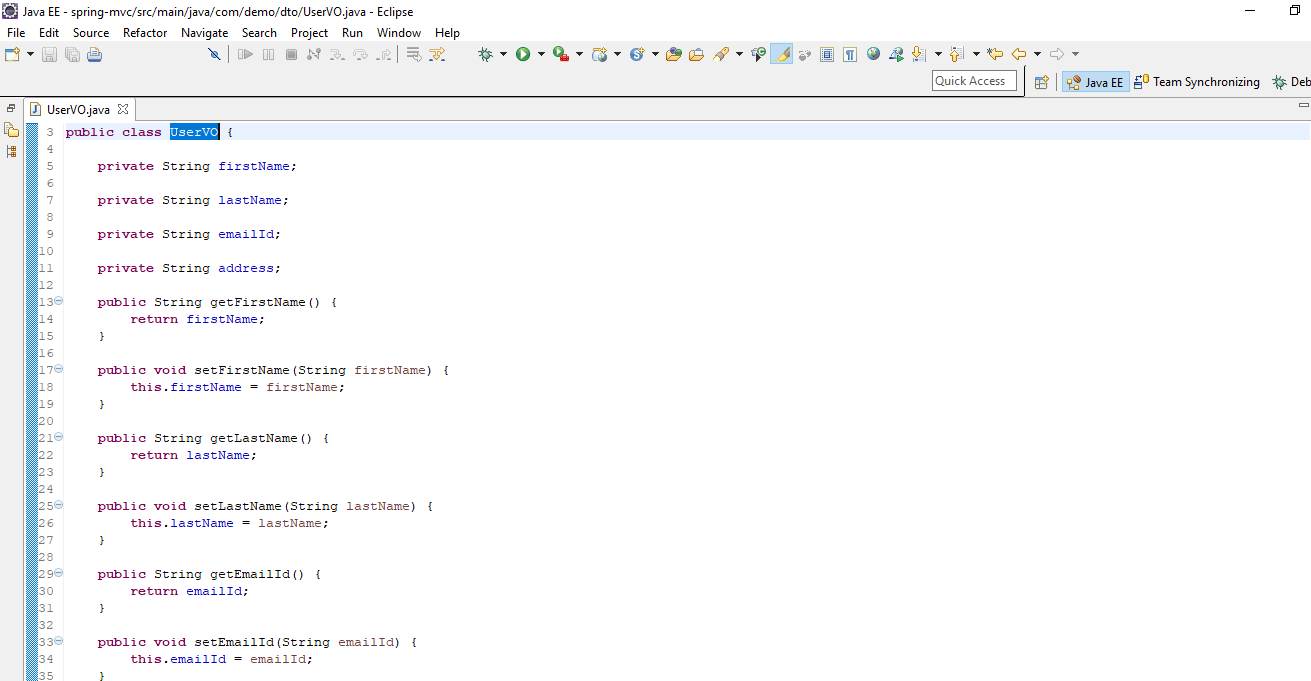


Output



**Form binding with command**

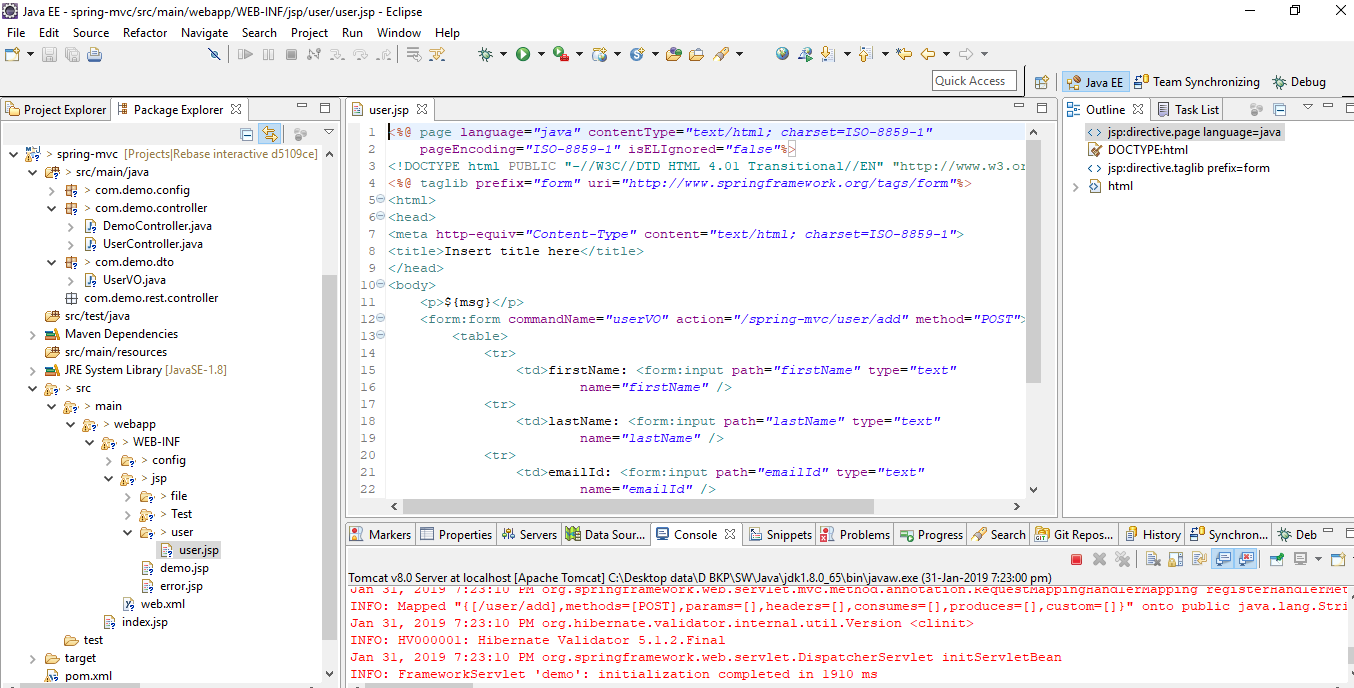


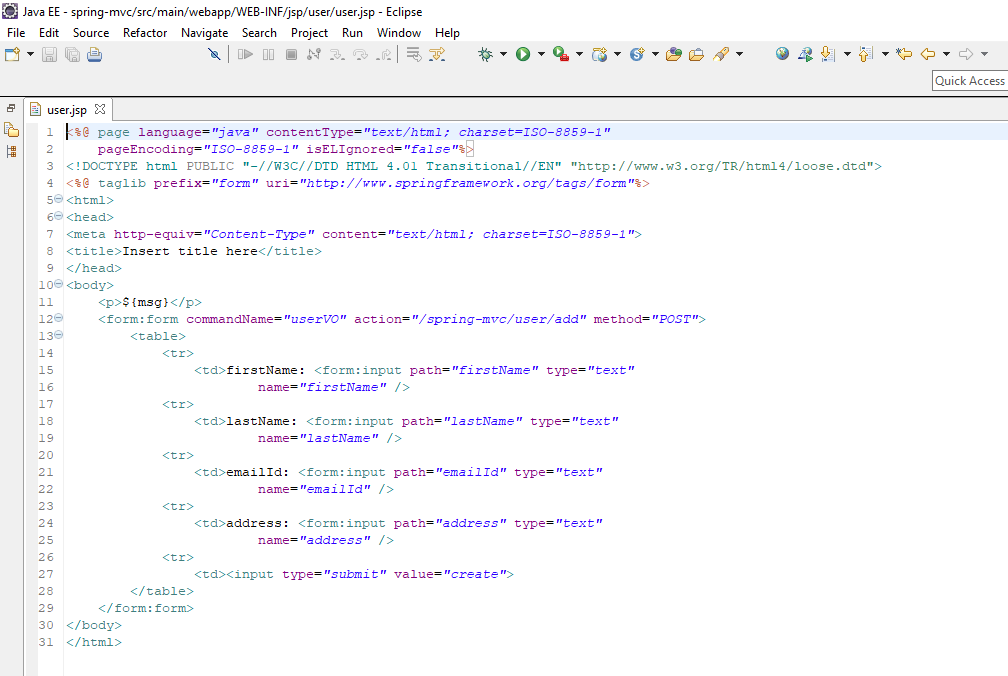


JSP (@ModelAttribute Name must match commandName)

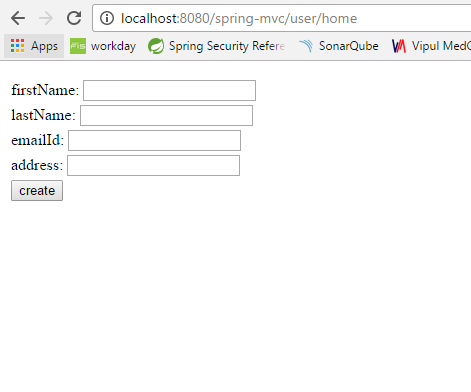
Include spring tld to create form tags

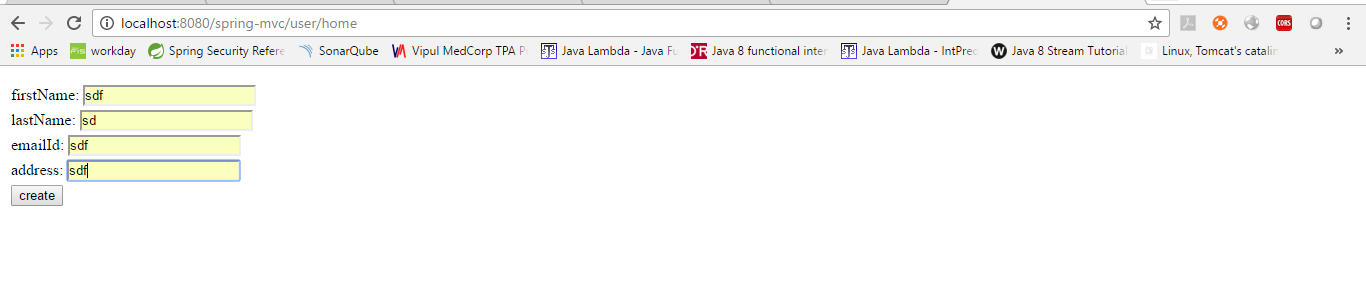
Name in path must match properties in modelAttribute POJO class



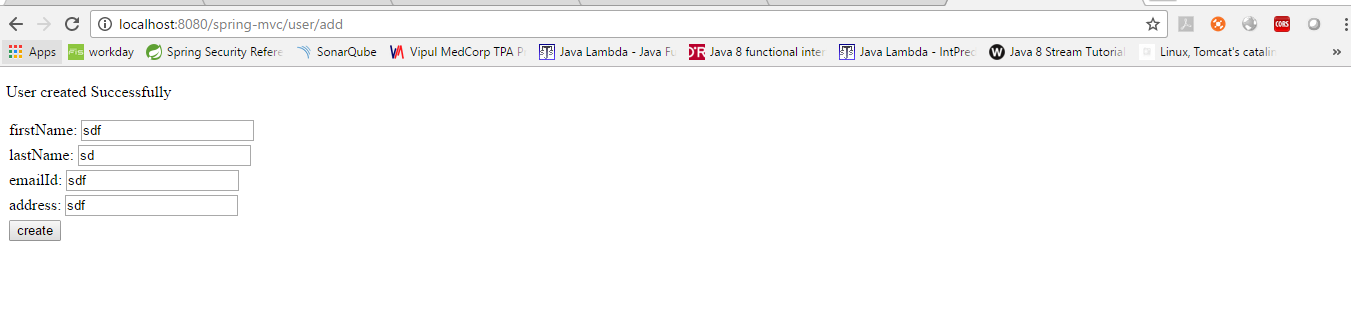


Run





Click create

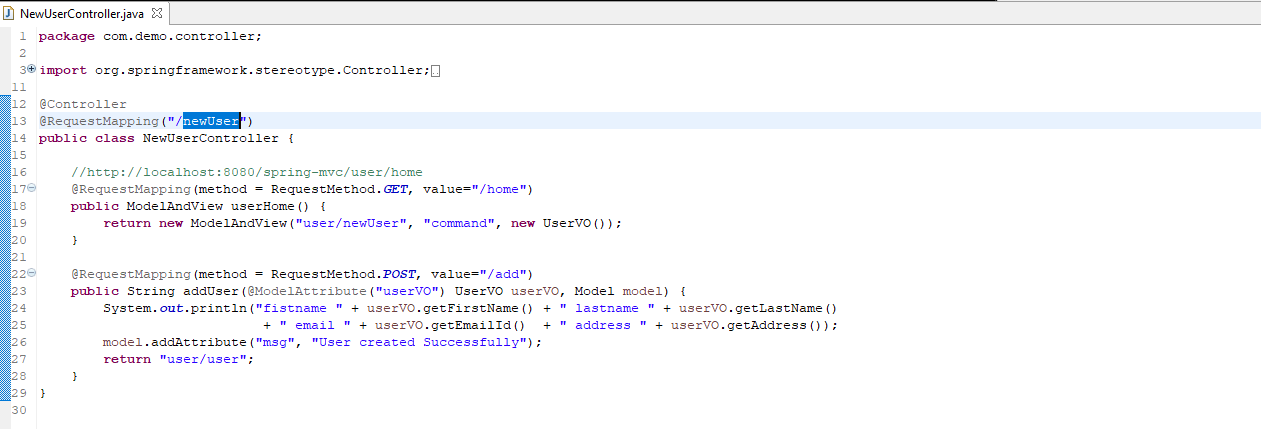


Check console

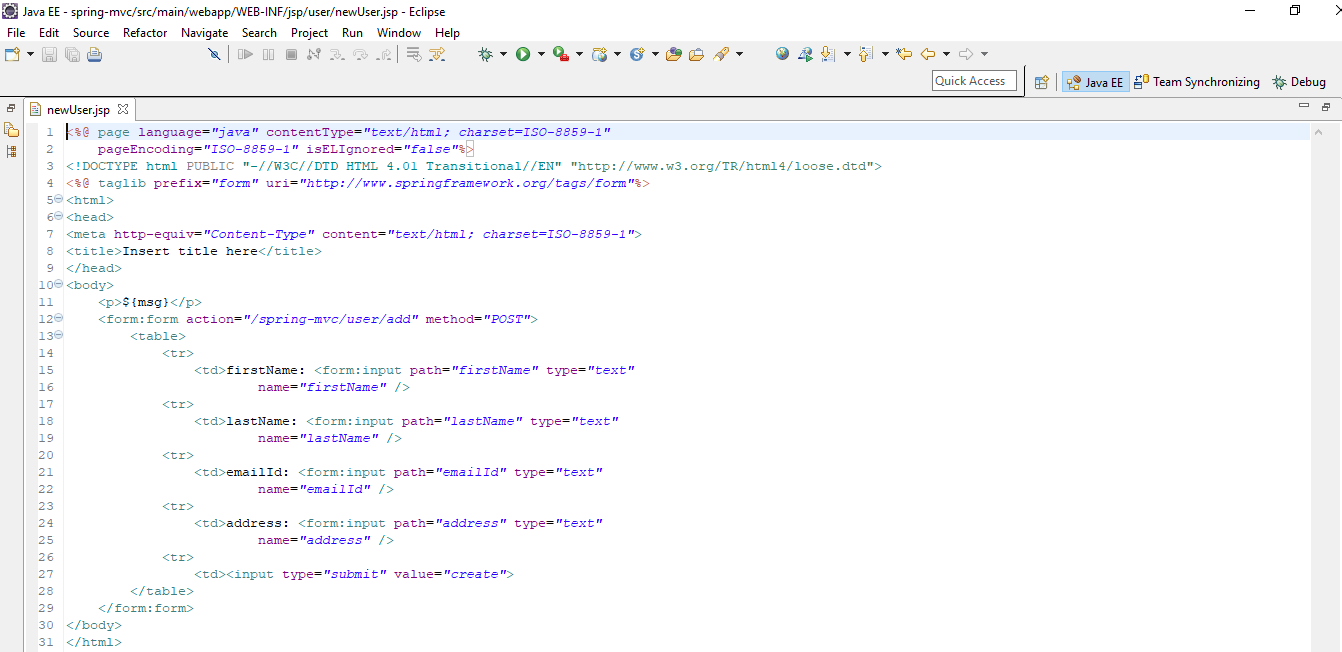


**Form binding without command**

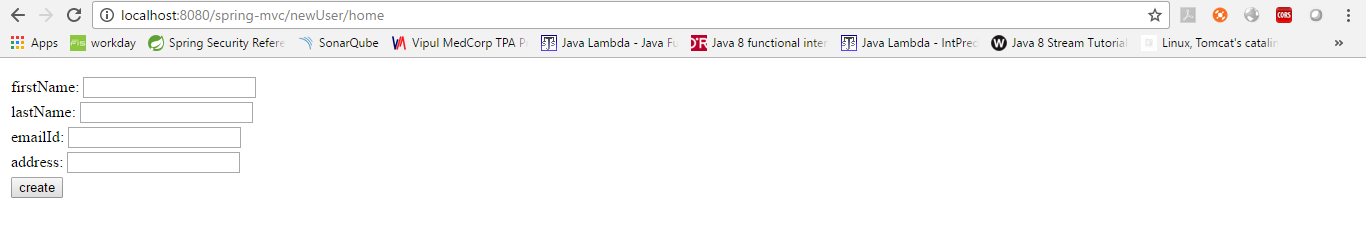
User ModelAndView to return modelName as command

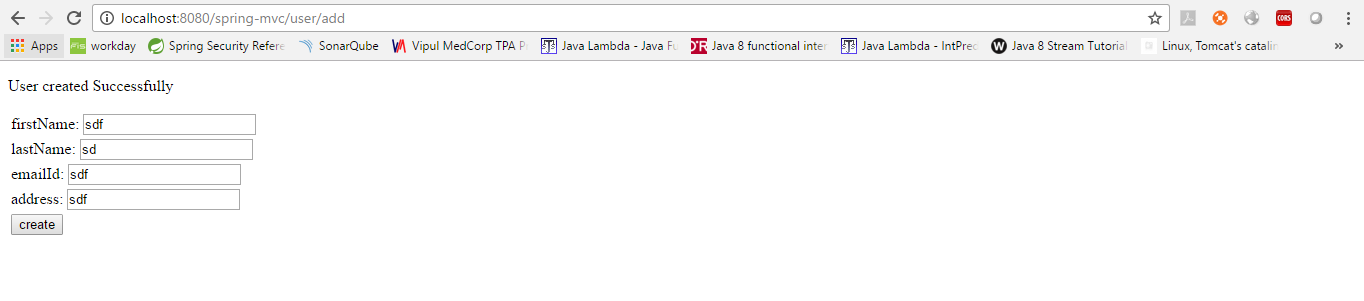


JSP (no commandName)

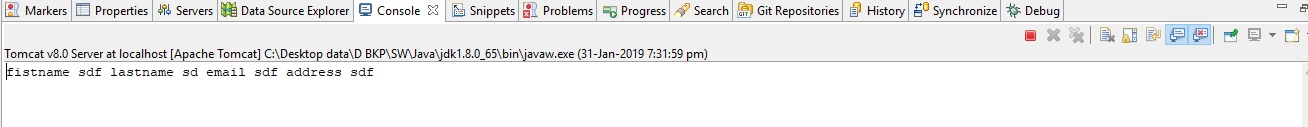


Run





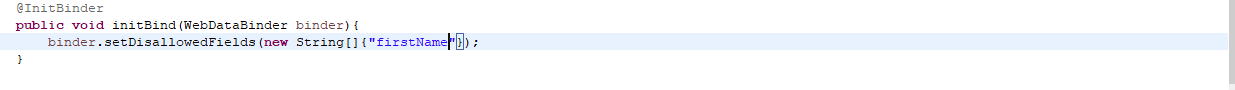
Console

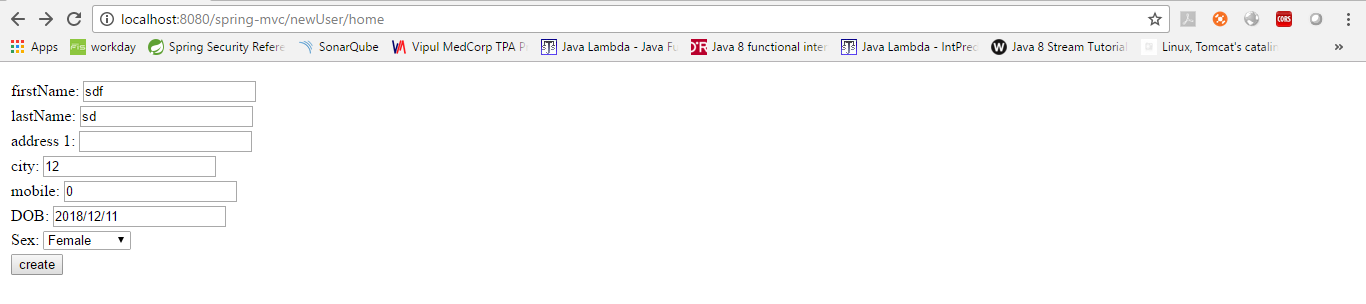


**MVC binding**

@InitBinder palys the role to identify the methods which initialize the WebDataBinder. The @InitBinder method supports many arguments as by @RequestMapping methods. @InitBinder method does not support command or form arguments. Mostly used argument is WebDataBinder in @InitBinder method.Example:

Not bind certain fields

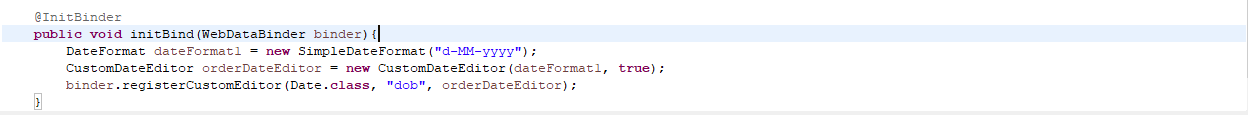




Click create

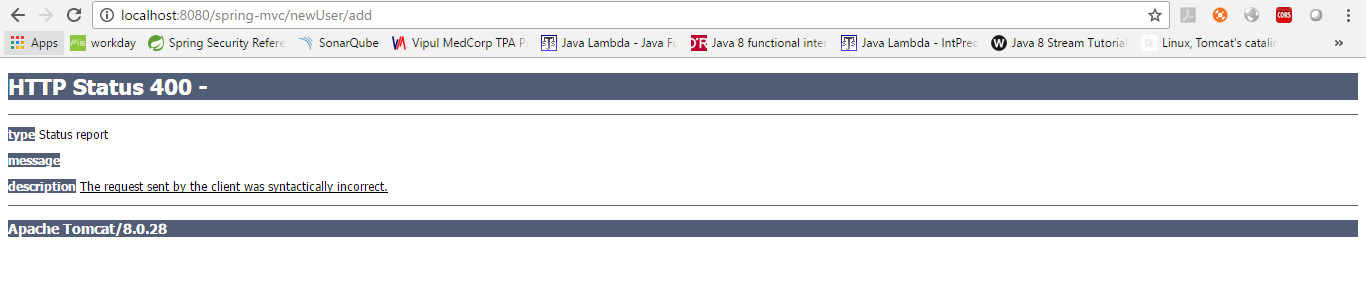


**Binding Custom formats**

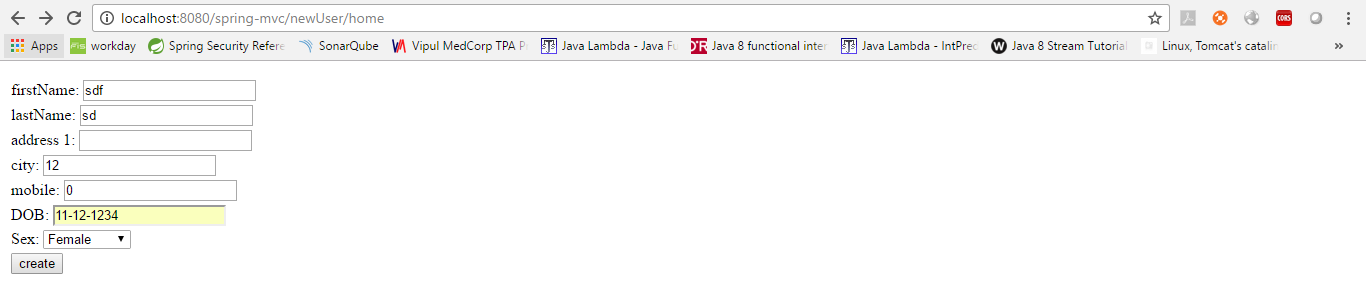


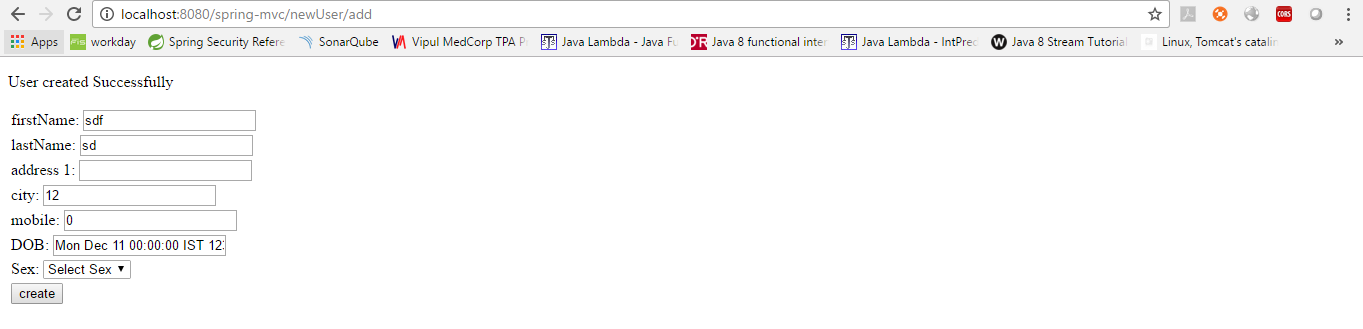


Click create



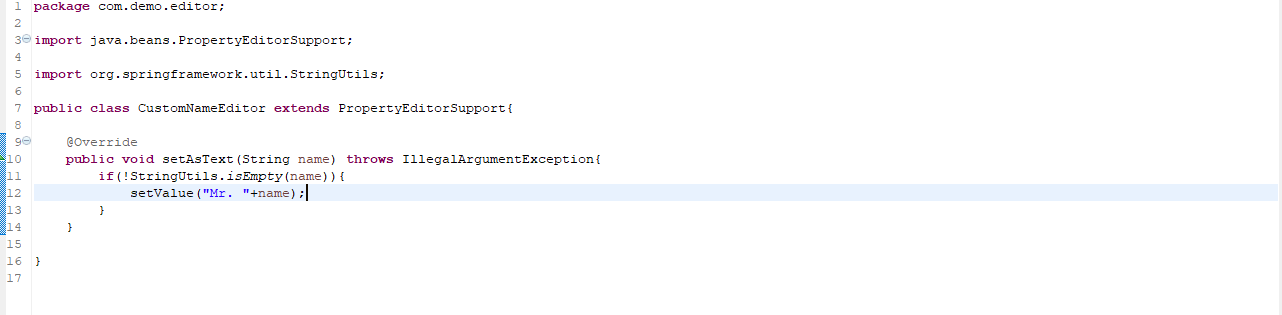
Send with valid date



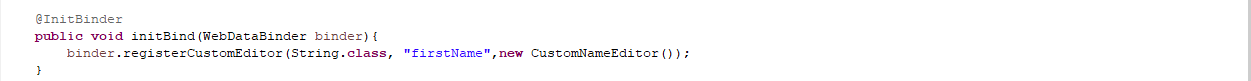


**Modify value before binding**

Create custom editor

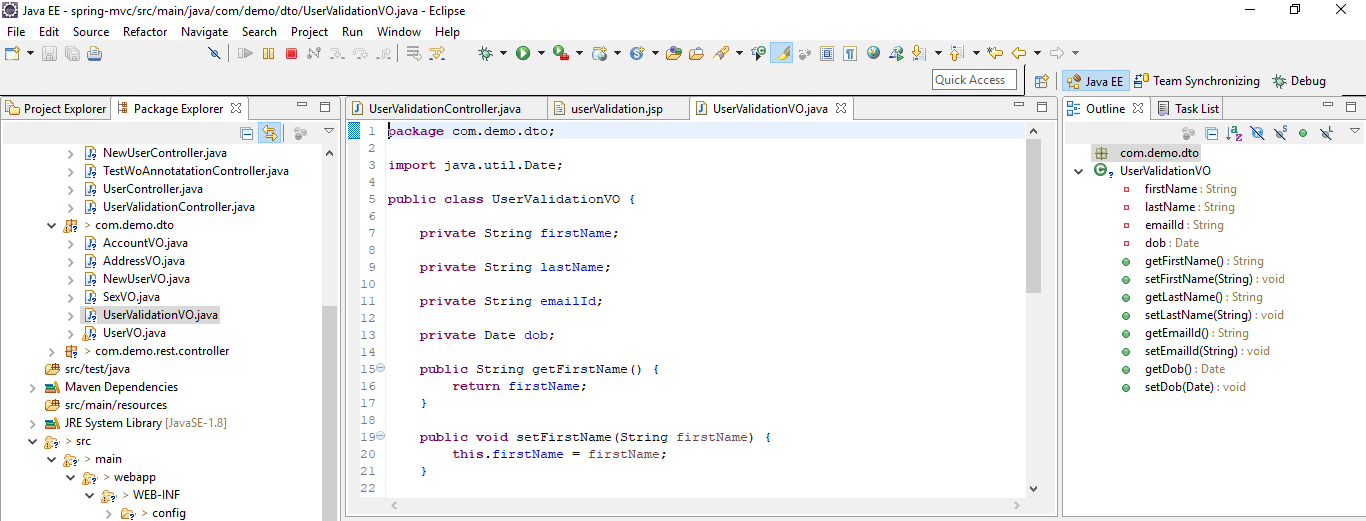


Use this in init bind

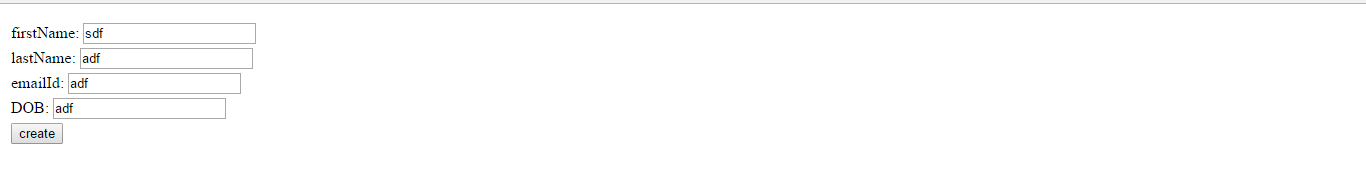


**MVC binding Validation**

Create Controller, view and Model for this



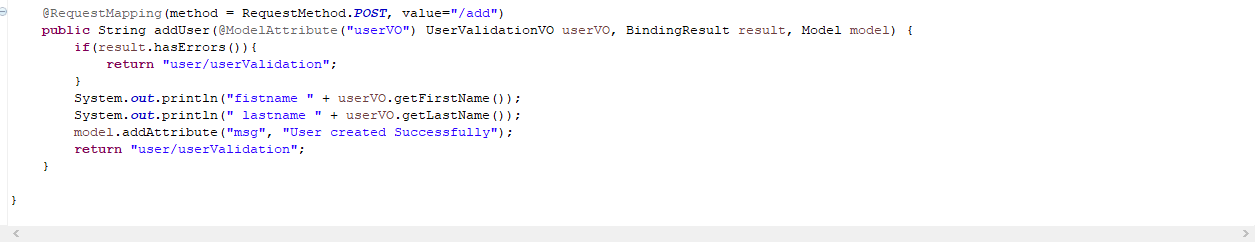
Enter invalid date of birth





To fix binding issues, add BindingResult in Controller and form:errors in jsp

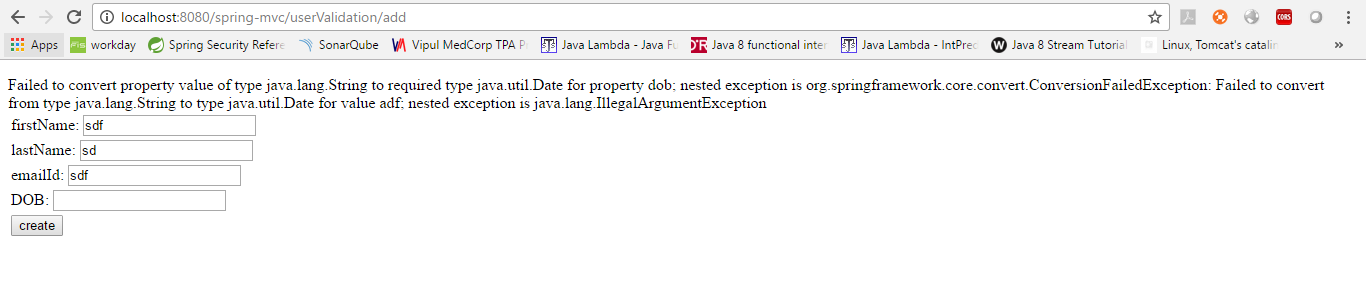
BindingResult should be before model parameter in method



Path=”\*”

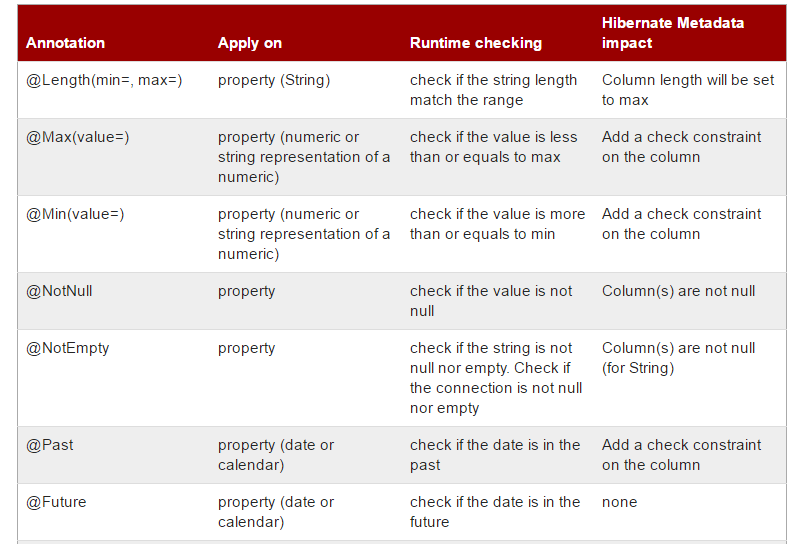




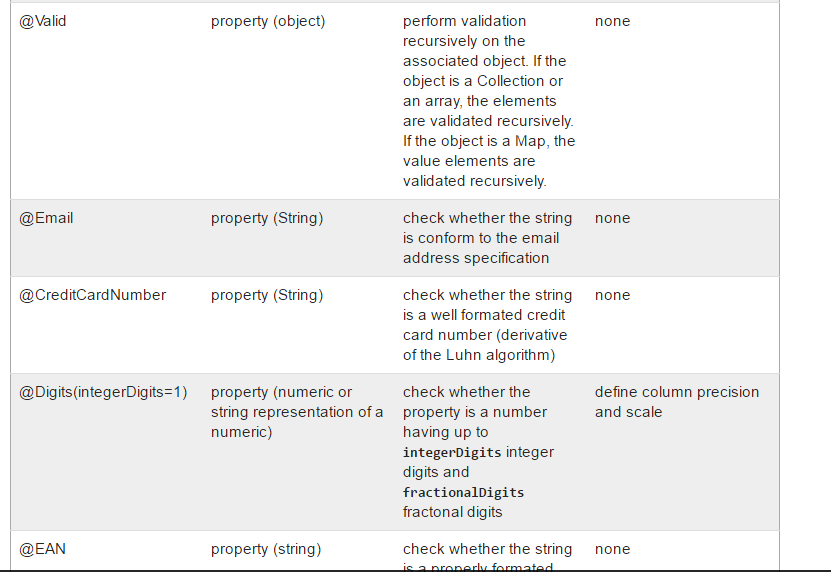


**VALIDATION WITH HIBERNATE VALIDATOR**

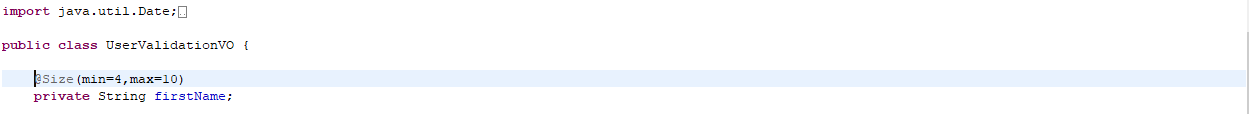
* DecimalMax
* DecimalMin
* Digits
* Email
* Future
* FutureOrPresent
* Max
* Min
* Negative
* NegativeOrZero
* NotBlank
* NotEmpty
* NotNull
* Null
* Past
* PastOrPresent
* Pattern
* Positive
* PositiveOrZero



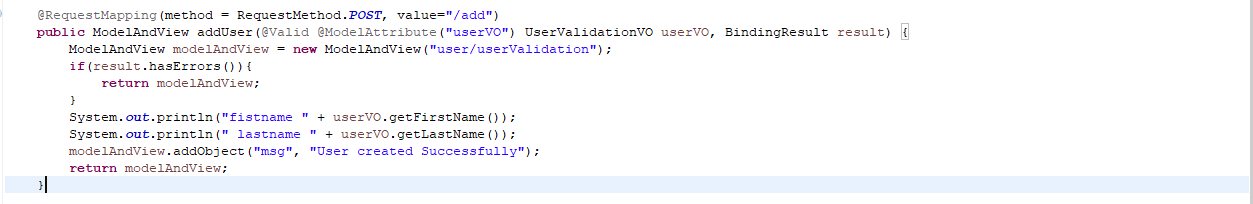


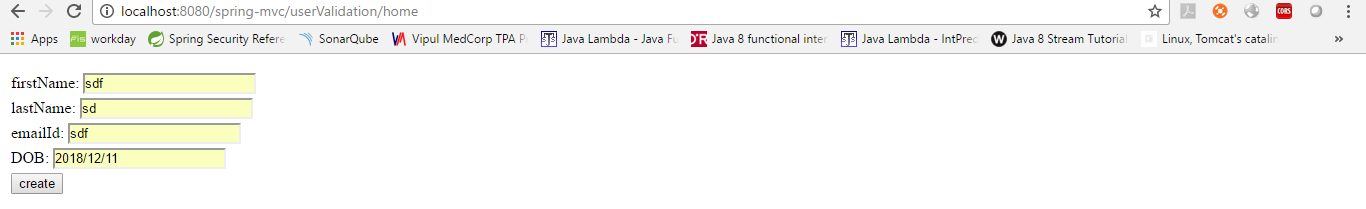


Define an annotation in model



Mark the controller method param with @Valid annotation







**CUSTOM ERROR MESSAGE**



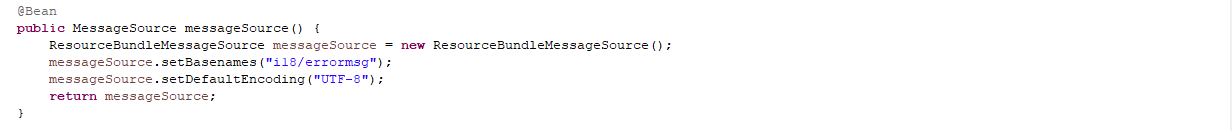
Click create

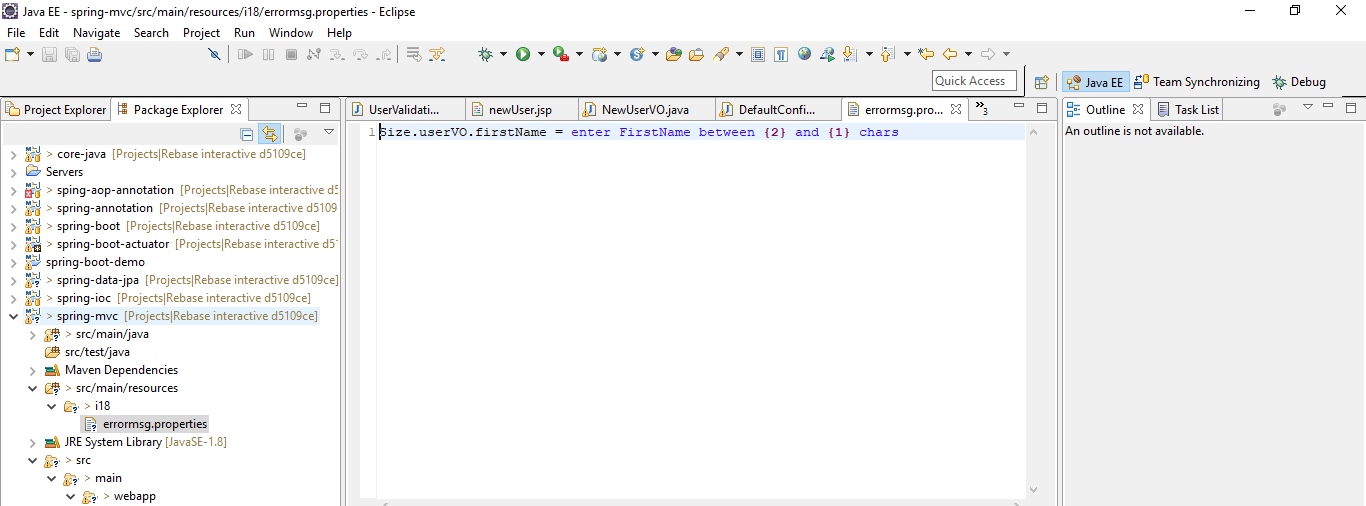


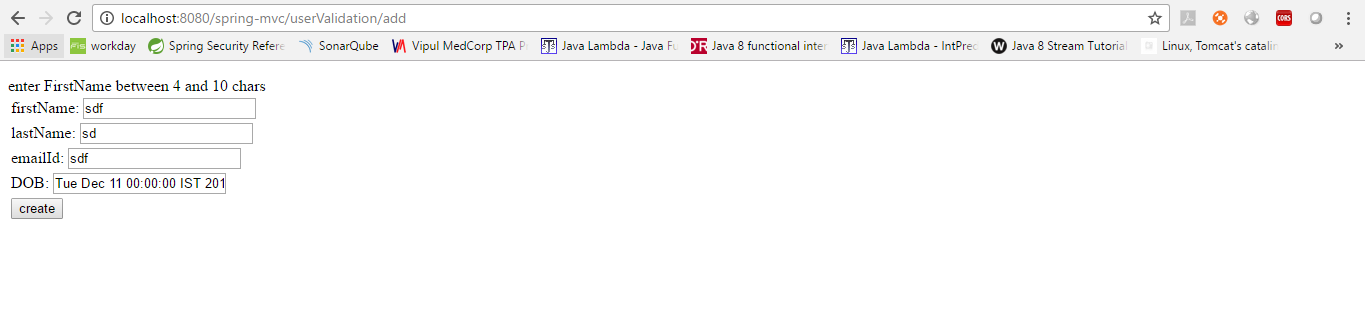
**CUSTOM ERROR MESSAGE WITH PROPERTY FILE**

create a properties with “key” and message. To know which @annotation bind to which key, just debug it and view value inside “BindingResult result“. Normally, the key is “@Annotation Name.object.fieldname“

create a message source bean

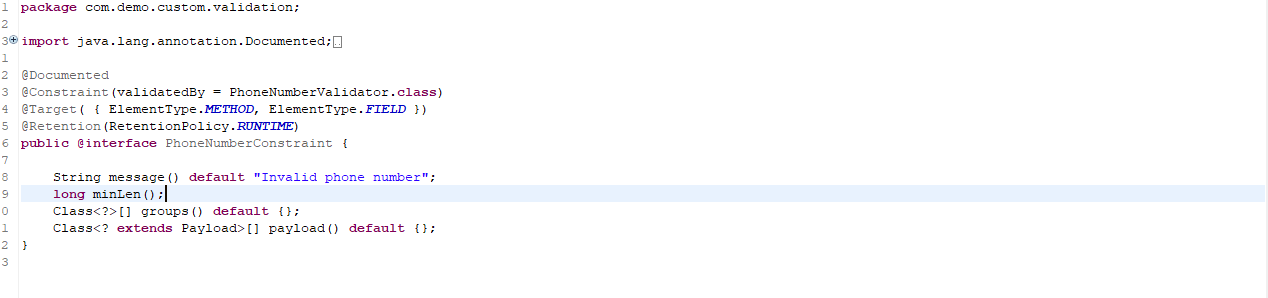




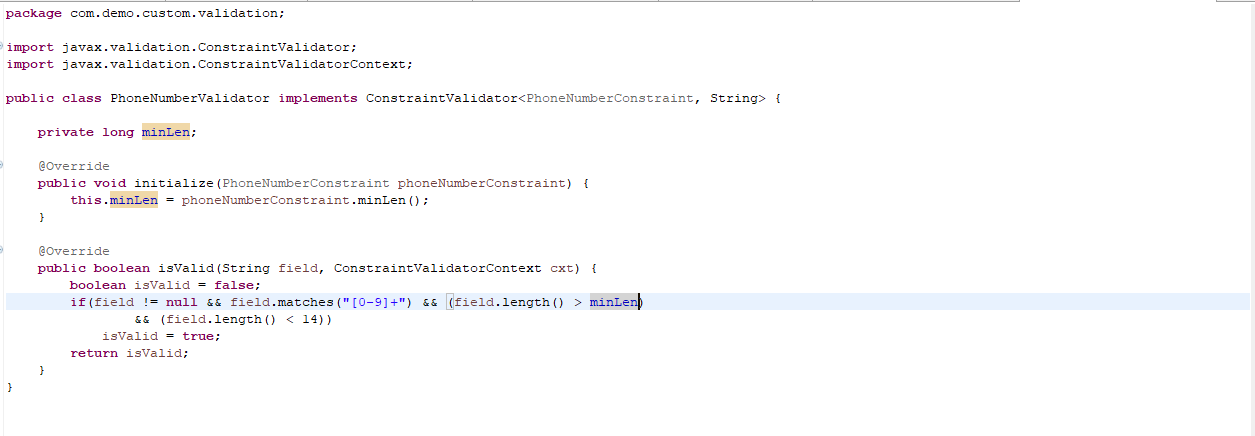


**CUSTOM ANNOTATION FOR VALIDATION**

Create custom annotation class



Create custom validator



Use the annotation



**MVC Validate Method**

Create validator class



Call this in controller



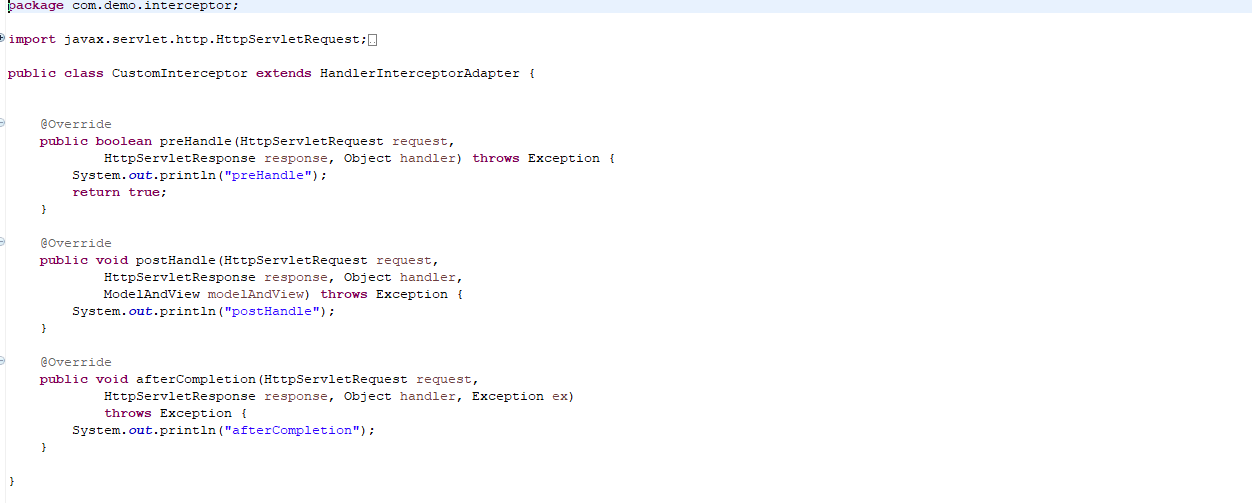
**FILTERS**

Configure a filter



**INTERCEPTORS**

Configure an interceptor

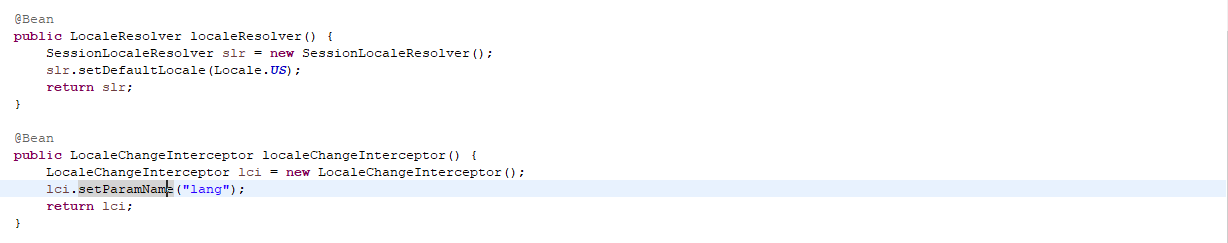


Register this interceptor



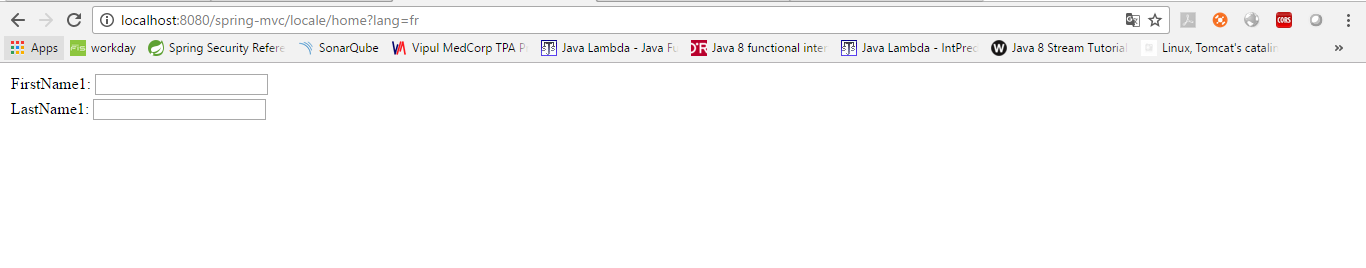
**INTERNALIZATION**

Configure interceptor and resolver

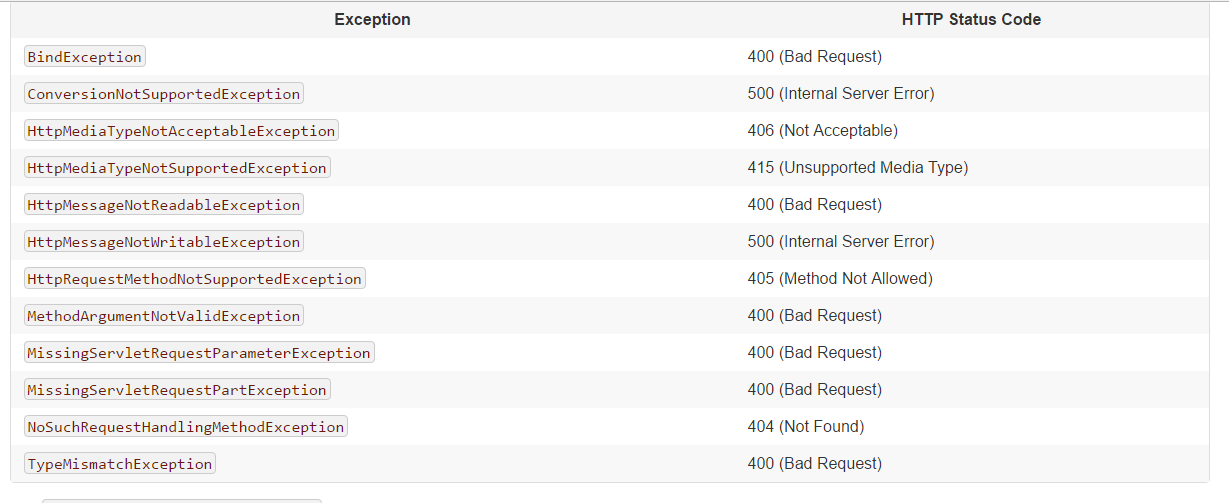


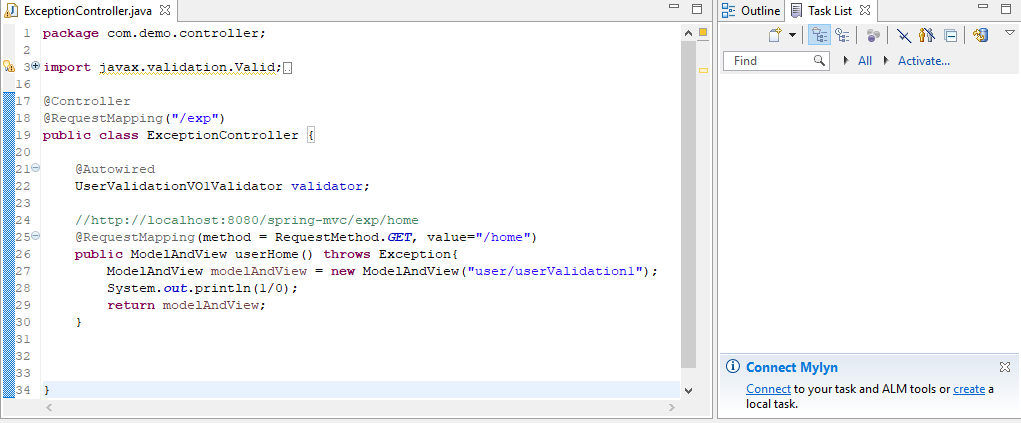
Test



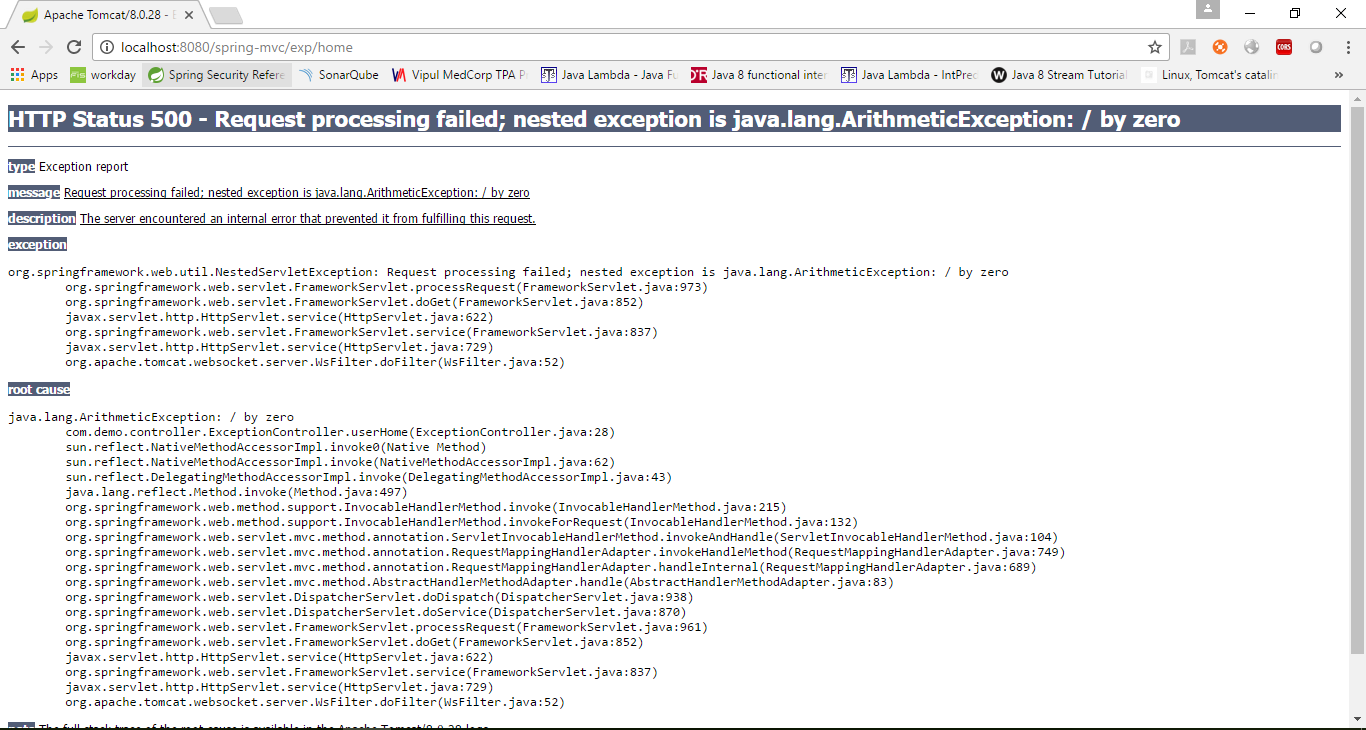


**Exception Handling**



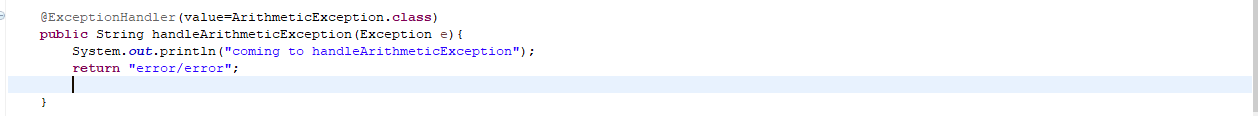


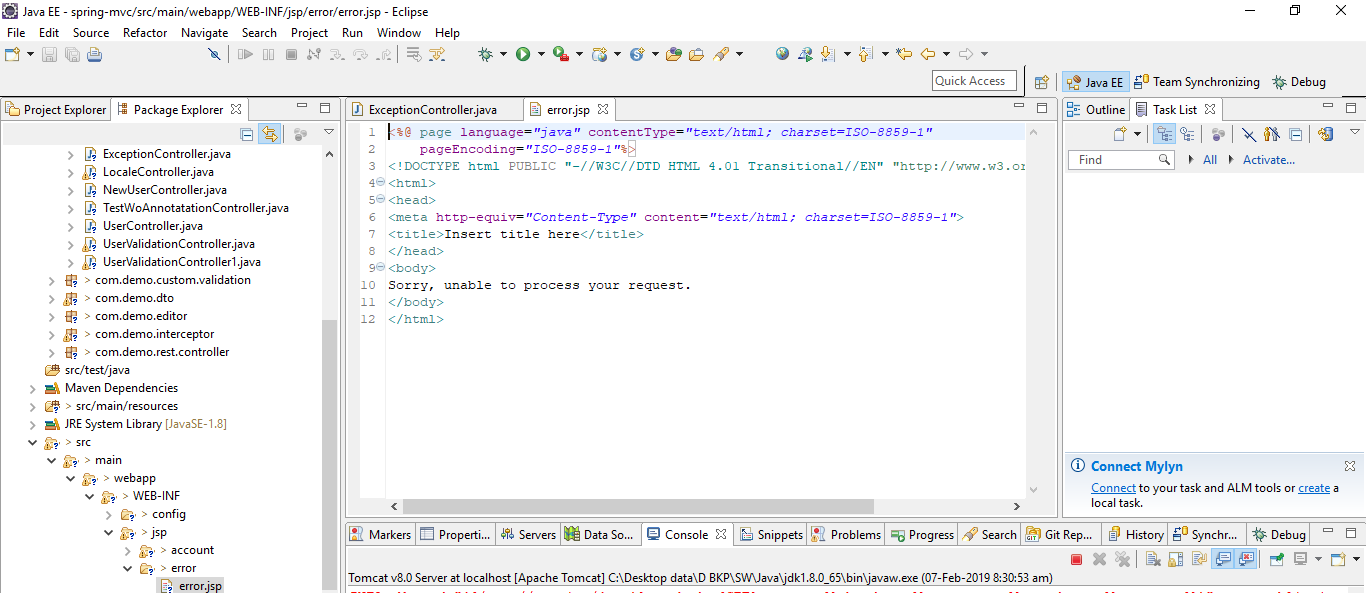
Run



**Custom Error Page**

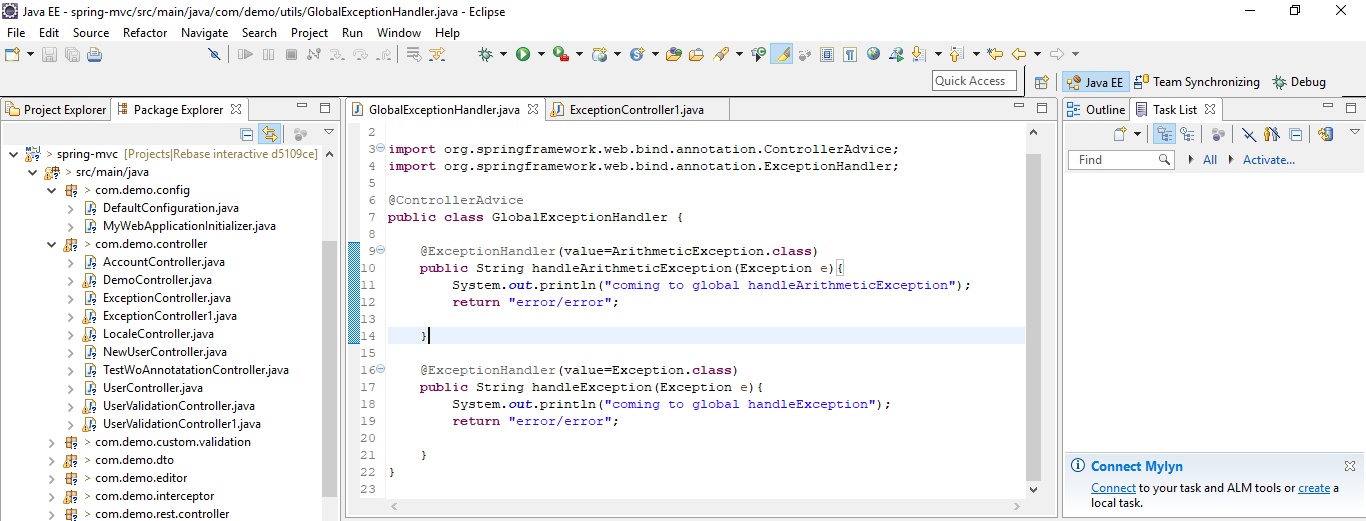
Use @ExceptionHandler



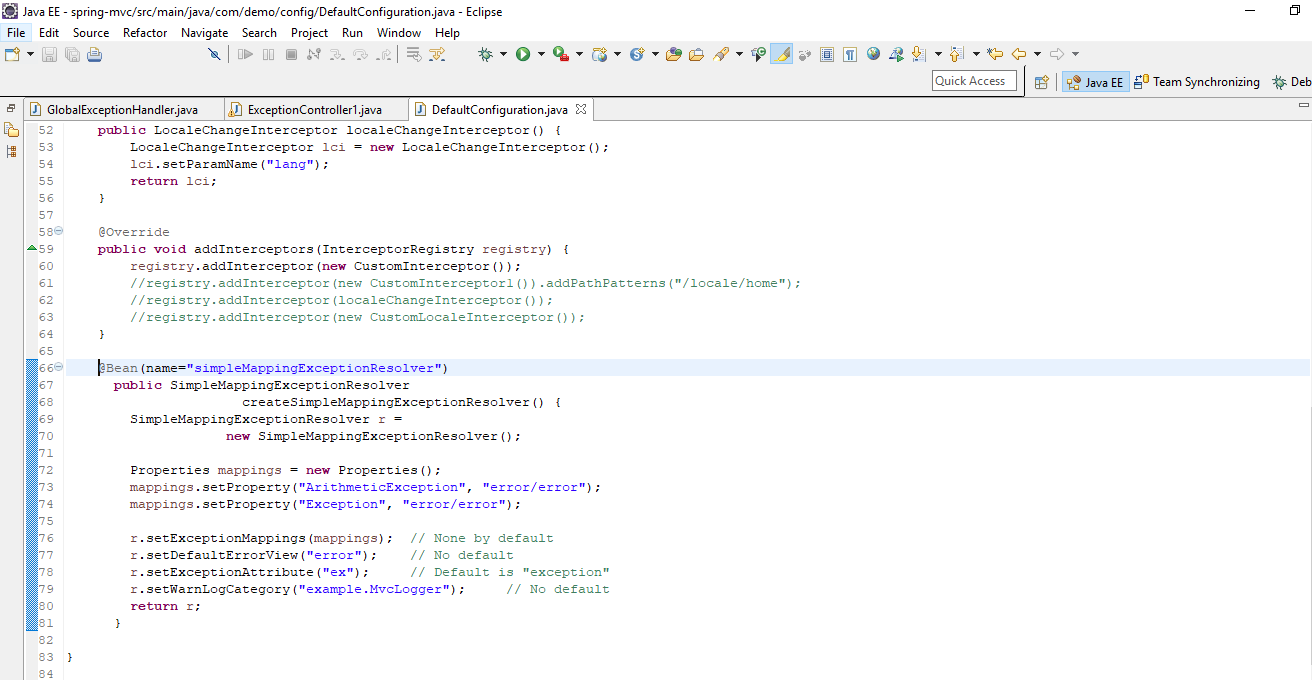




**Common Exception Handler**

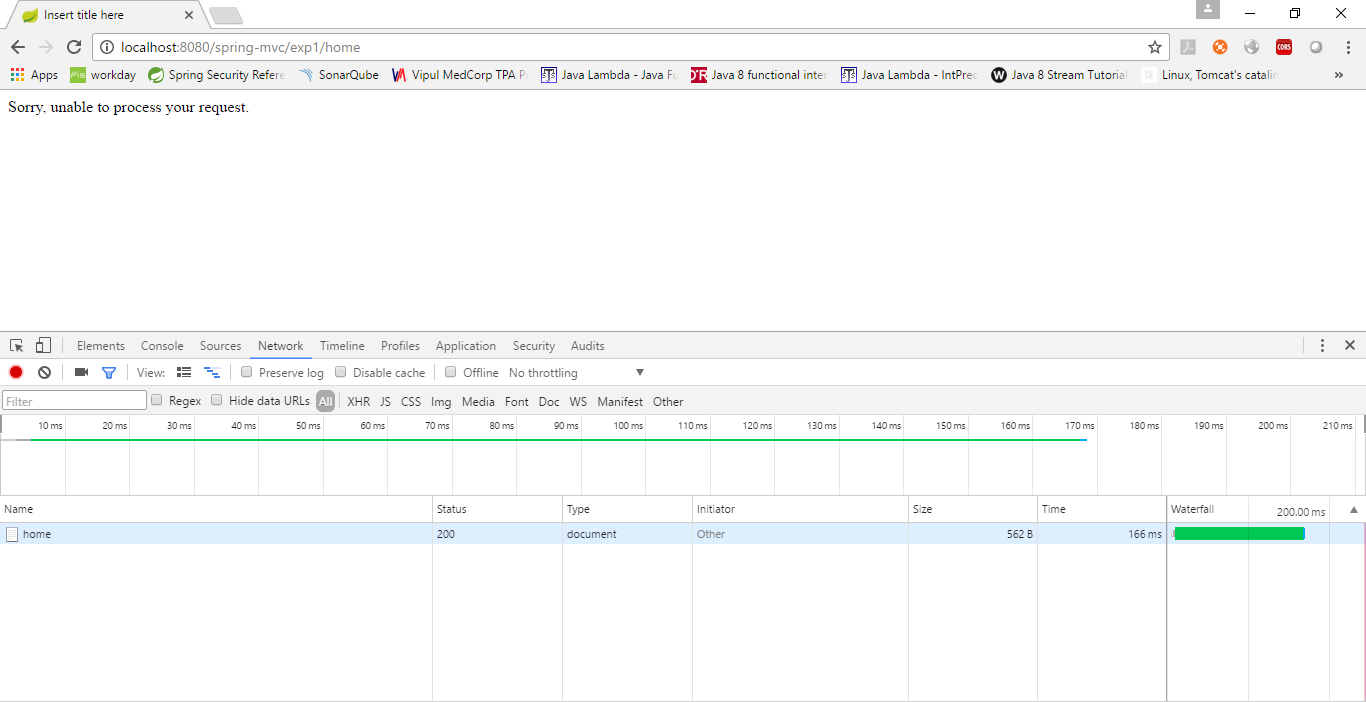


**ExceptionResolver**

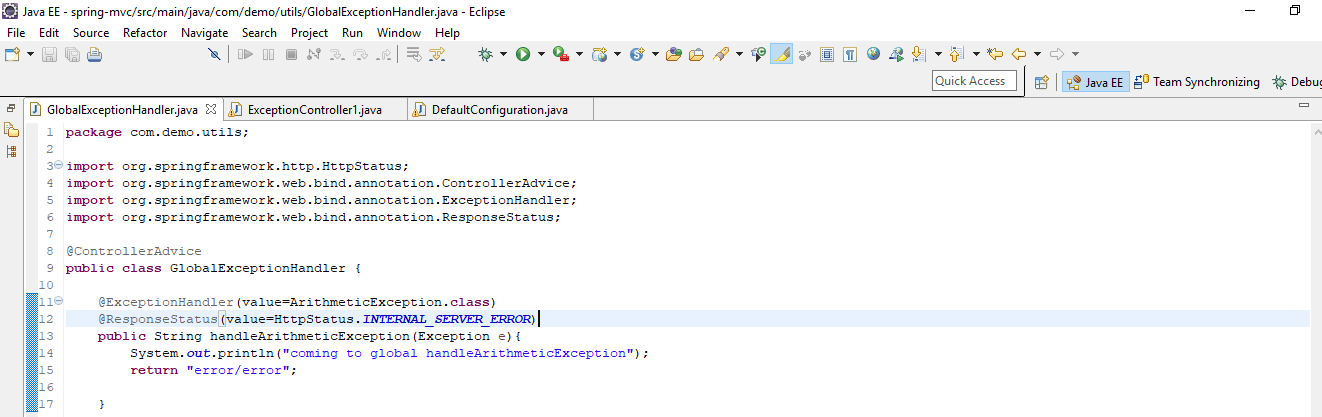


**EXCEPTION STATUS CODES**

By default, server send 200 Ok for exceptions



To send error status codes use annotation @ResponseStatus



**Redirect**

Browser URL will change to redirected URL

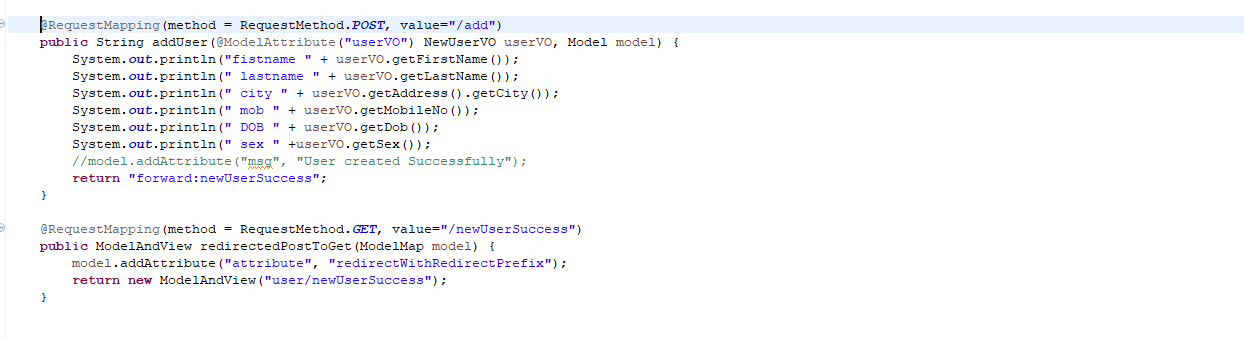
When a view name is returned with the prefix redirect: –theUrlBasedViewResolver (and all its subclasses) will recognize this as a special indication that a redirect needs to happen. The rest of the view name will be used as the redirect URL.

redirect will respond with a 302 and the new URL in the Location header; the browser/client will then make another request to the new URL



**Forward**

forward happens entirely on a server side; the Servlet container forwards the same request to the target URL; the URL won’t change in the browser



**RedirectAttributes**

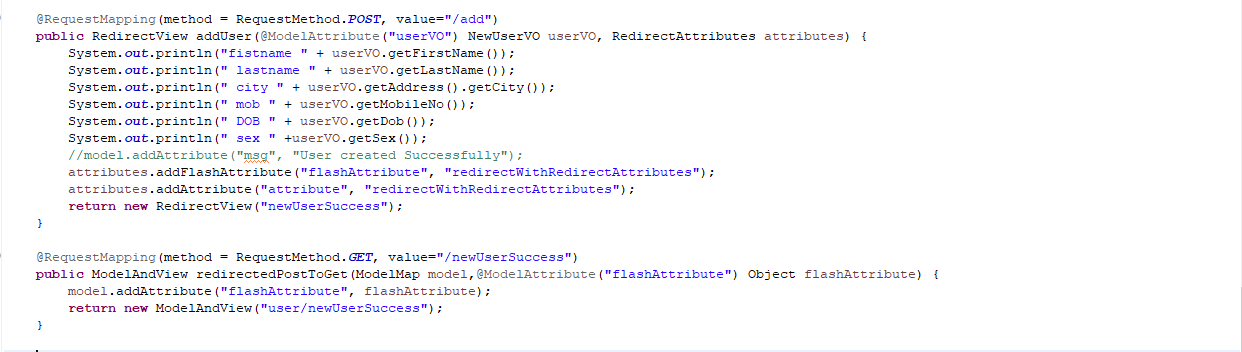
RedirectView will trigger a HttpServletResponse.sendRedirect() – which will perform the actual redirect.

Notice here how we’re injecting the redirect attributes into the method – the framework will do the heavy lifting here and allow us to interact with these attributes.

We’re adding the model attribute attribute – which will be exposed as HTTP query parameter. The model must contain only objects – generally Strings or objects that can be converted to Strings.

we can inject the attributes object in the method directly – which makes this mechanism very easy to use.

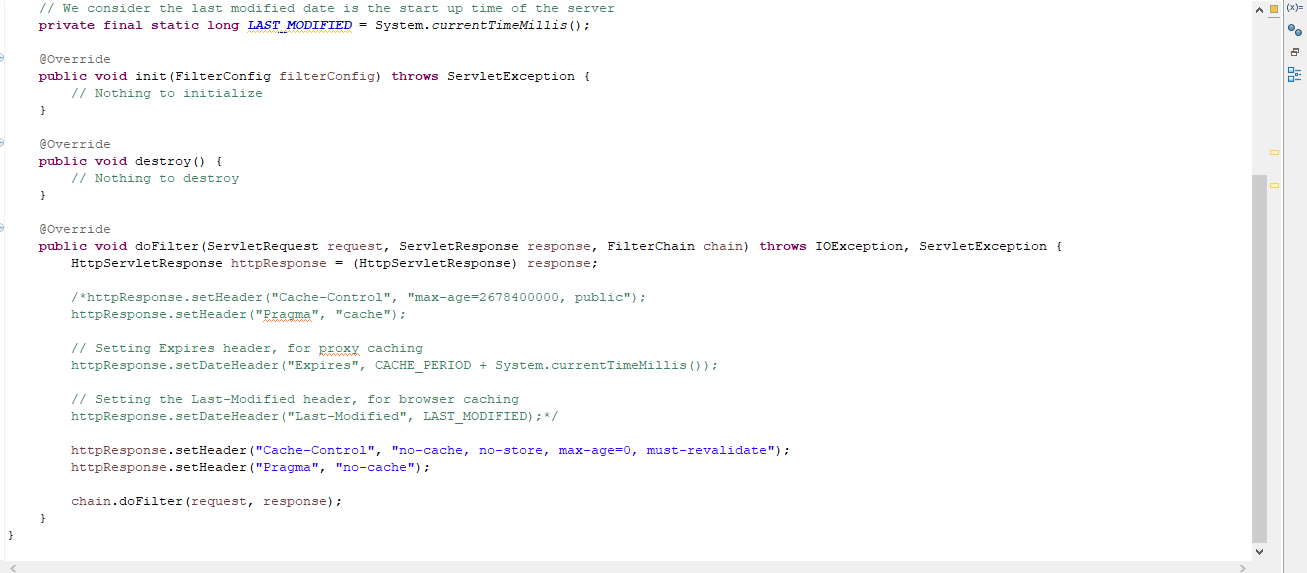
Notice also that we are adding a flash attribute as well – this is an attribute that won’t make it into the URL. What we can achieve with this kind of attribute is – we can later access the flash attribute using @ModelAttribute(“flashAttribute”) only in the method that is the final target of the redirect:



**SPRING REST**

**BASICS**

**Http GET**

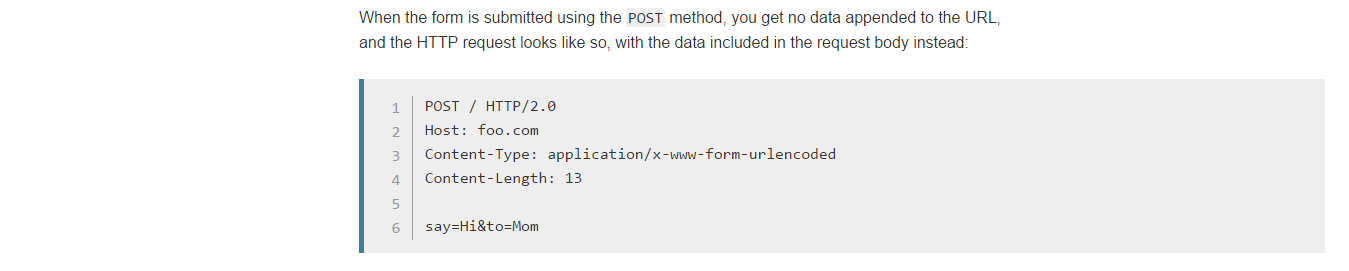


**Http POST**

The POST method is a little different. It's the method the browser uses to talk to the server when asking for a response that takes into account the data provided in the body of the HTTP request: "Hey server, take a look at this data and send me back an appropriate result." If a form is sent using this method, the data is appended to the body of the HTTP request.

Let's look at an example — this is the same form we looked at in the GET section above, but with the method attribute set to post.





**Http Headers**

HTTP headers allow the client and the server to pass additional information with the request or the response. An HTTP header consists of its case-insensitive name followed by a colon ':', then by its value (without line breaks). Leading white space before the value is ignored.

Custom proprietary headers can be added using the 'X-' prefix

Headers can be grouped according to their contexts:

* General header: Headers applying to both requests and responses but with no relation to the data eventually transmitted in the body.
* Request header: Headers containing more information about the resource to be fetched or about the client itself.
* Response header: Headers with additional information about the response, like its location or about the server itself (name and version etc.).
* Entity header: Headers containing more information about the body of the entity, like its content length or its MIME-type.

Headers can also be grouped according to how proxies handle them:

**End-to-end headers**

These headers must be transmitted to the final recipient of the message; that is, the server for a request or the client for a response. Intermediate proxies must retransmit end-to-end headers unmodified and caches must store them.

**Hop-by-hop headers**

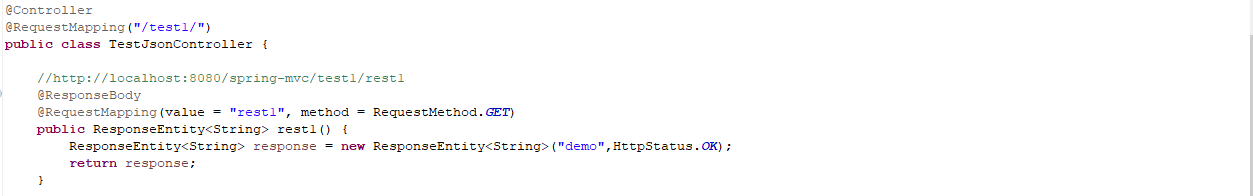
These headers are meaningful only for a single transport-level connection and must not be retransmitted by proxies or cached. Such headers are: Connection, Keep-Alive, Proxy-Authenticate, Proxy-Authorization, TE, Trailer, Transfer-Encoding and Upgrade. Note that only hop-by-hop headers may be set using the Connection general header.

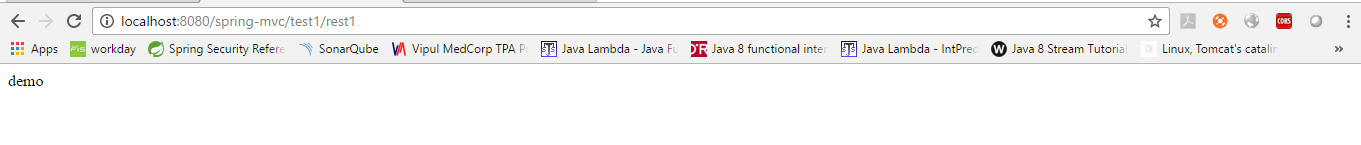
Check this link for more:

<https://en.wikipedia.org/wiki/List_of_HTTP_header_fields#Field_names>

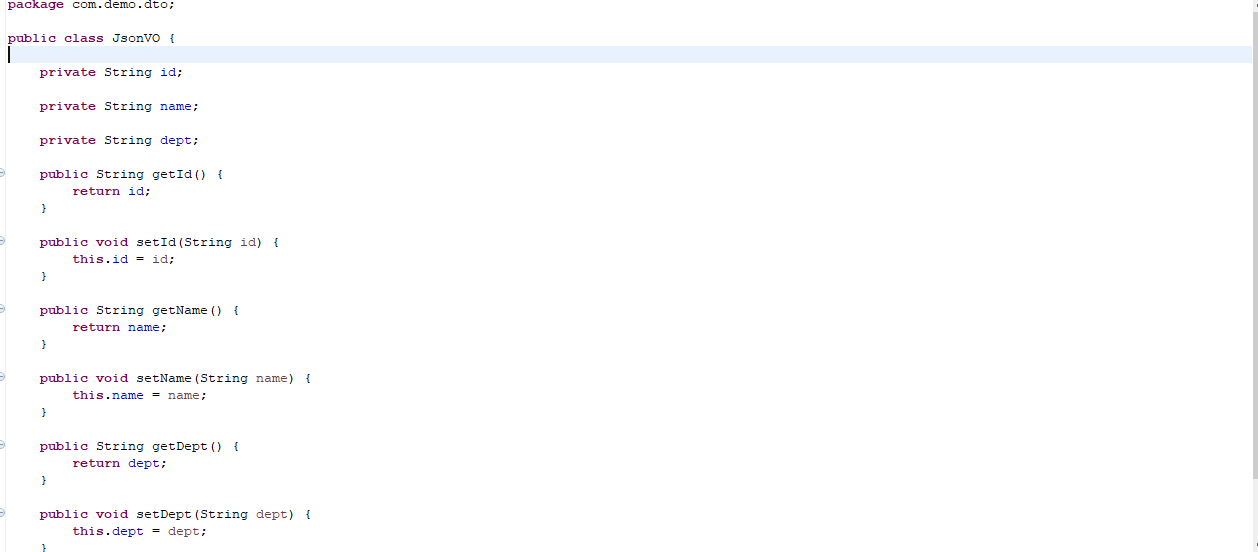
**JSON**

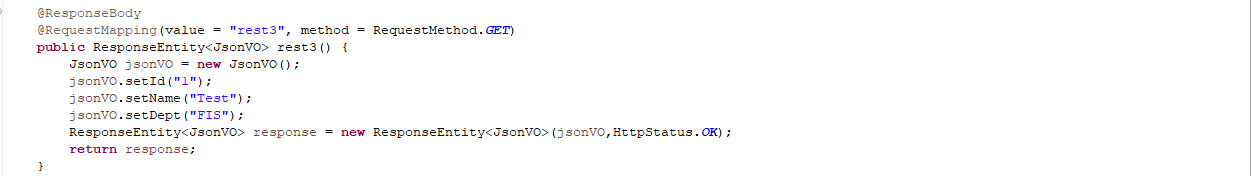
For Json, include Jackson jars





Test with Json. Pass Java PoJo class, spring will convert into json



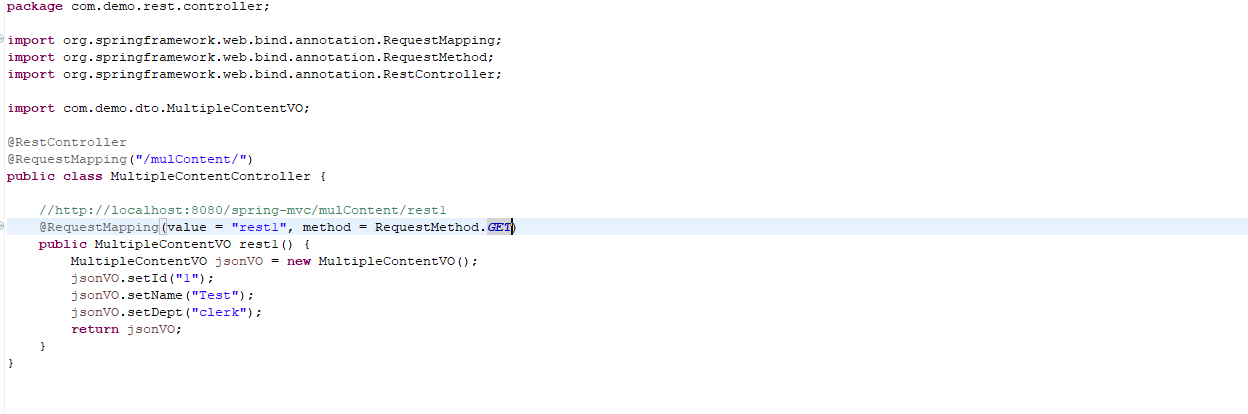


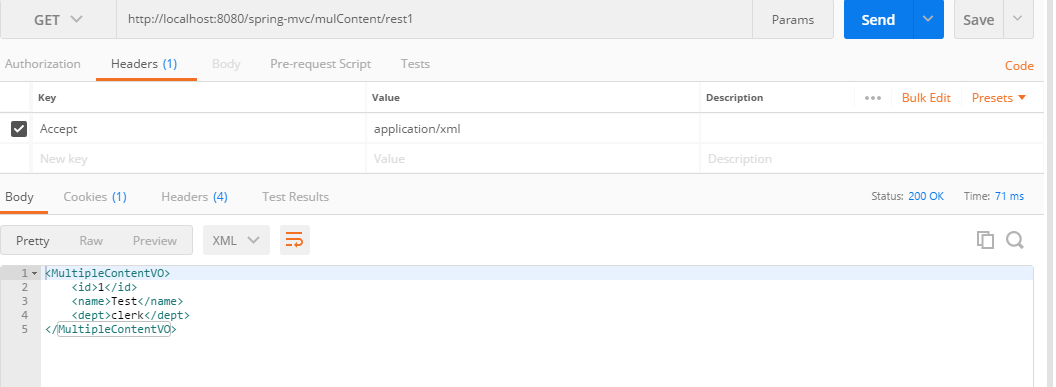


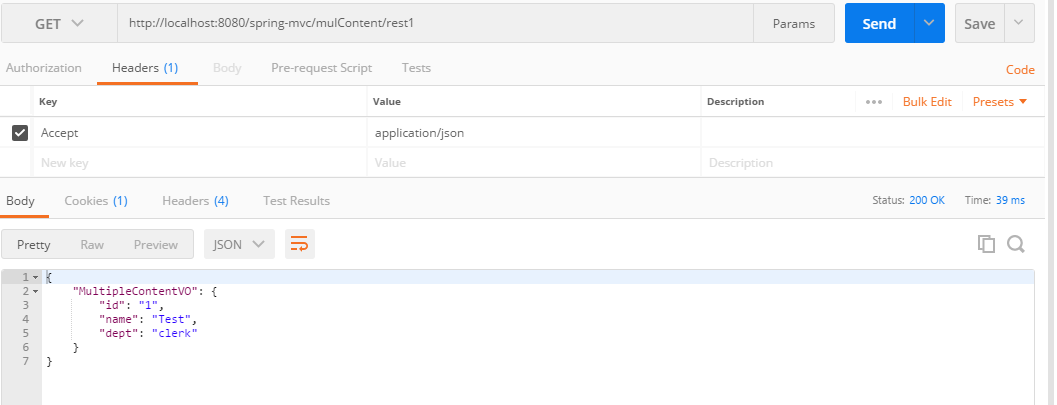
For XML include Jackson XML jars

**MULTIPLE FORMATS**

Now application includes both json and XML. Accept header will decide the response format

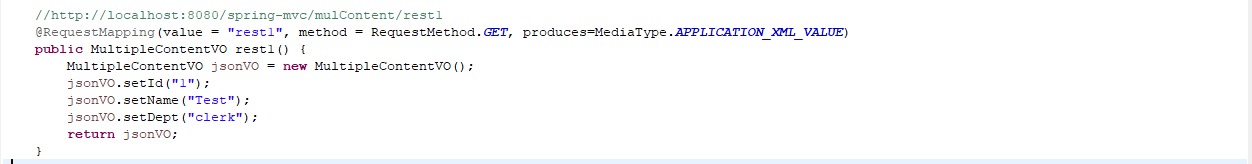


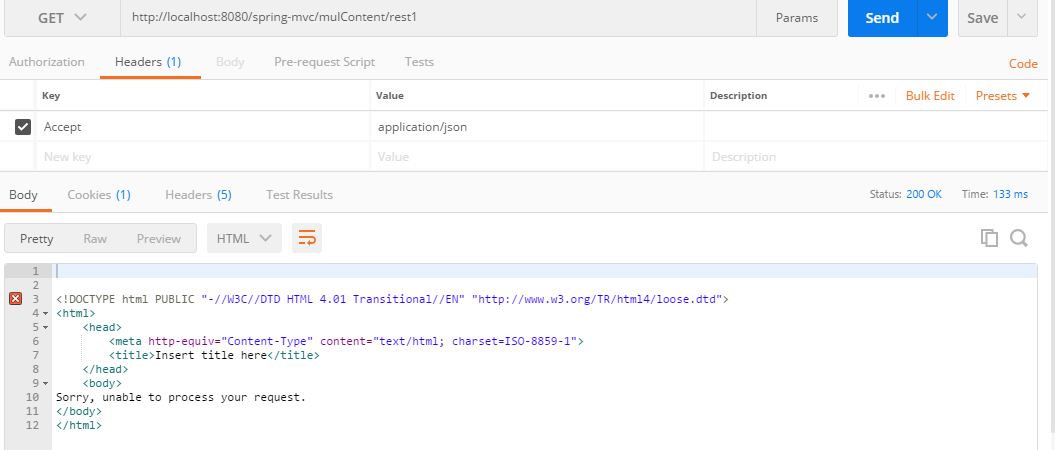


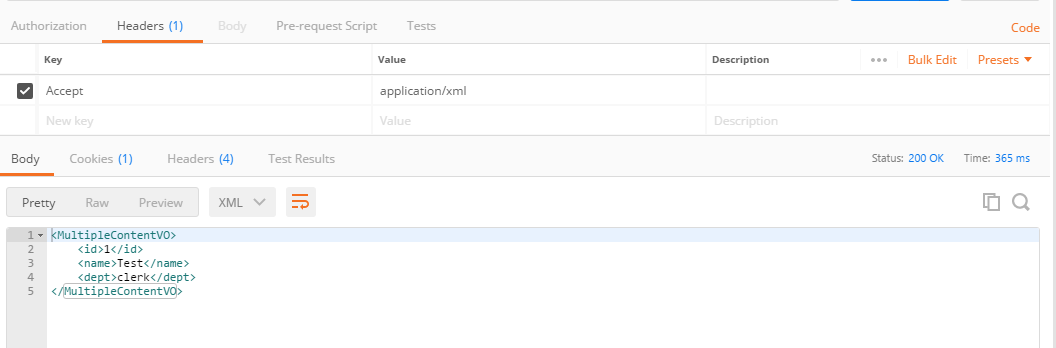


We can specify in server to restrict client from deciding

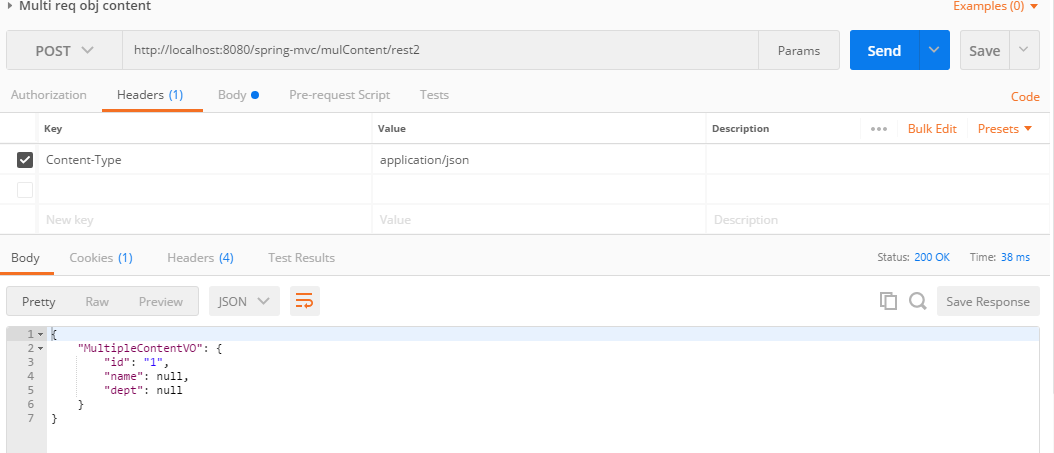
**@produces**

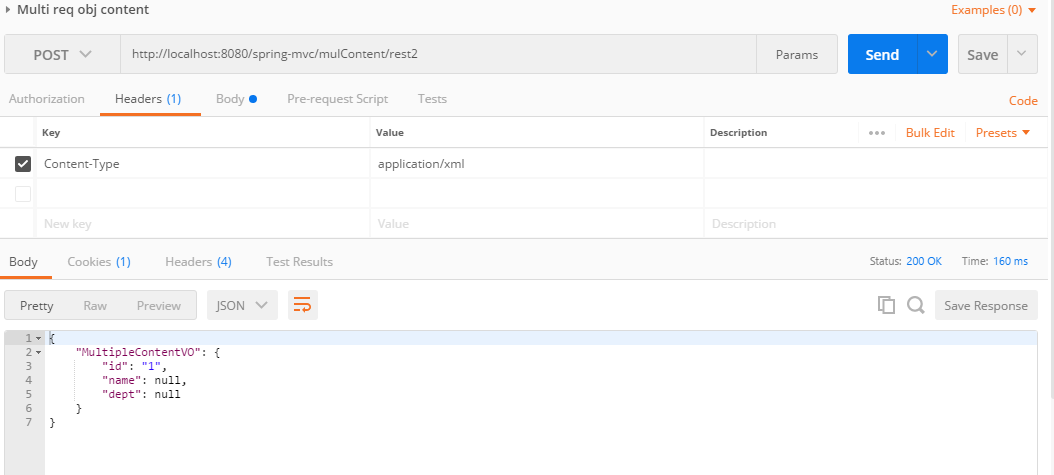


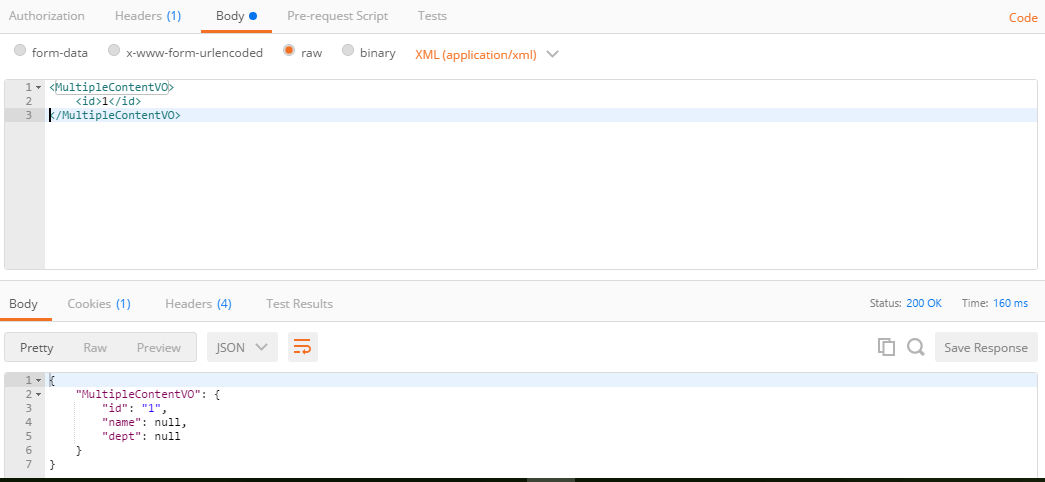




Now application includes both json and XML. Content-type header will decide the request format

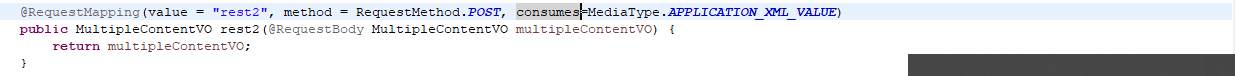






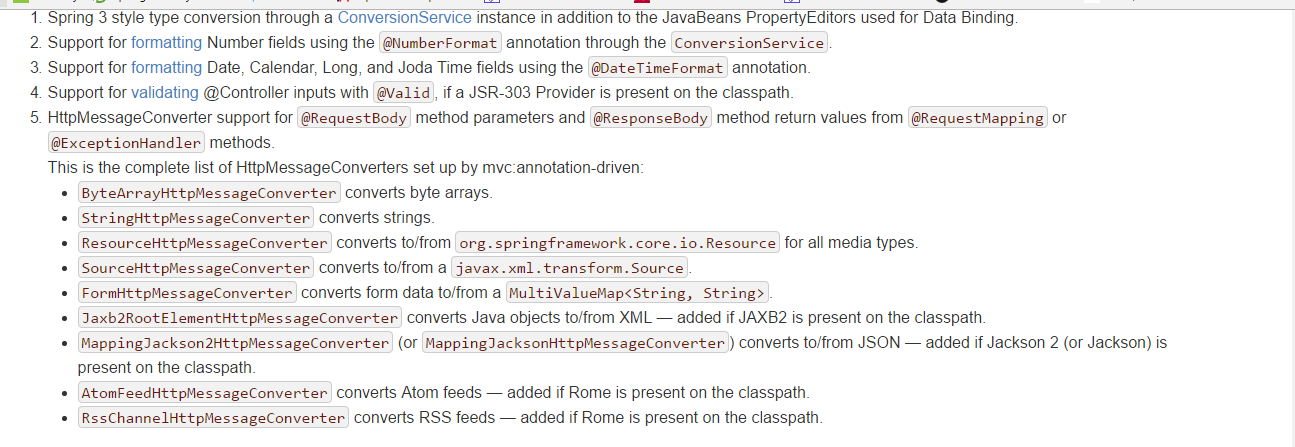
We can specify in server to restrict client from deciding

**@consumes**

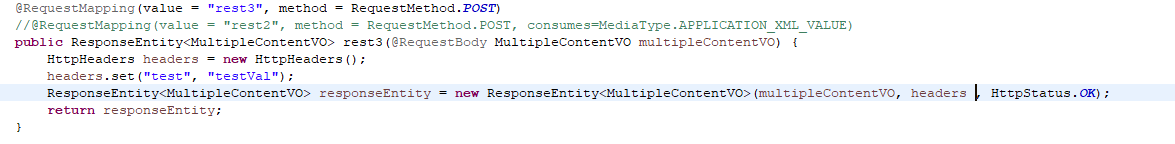


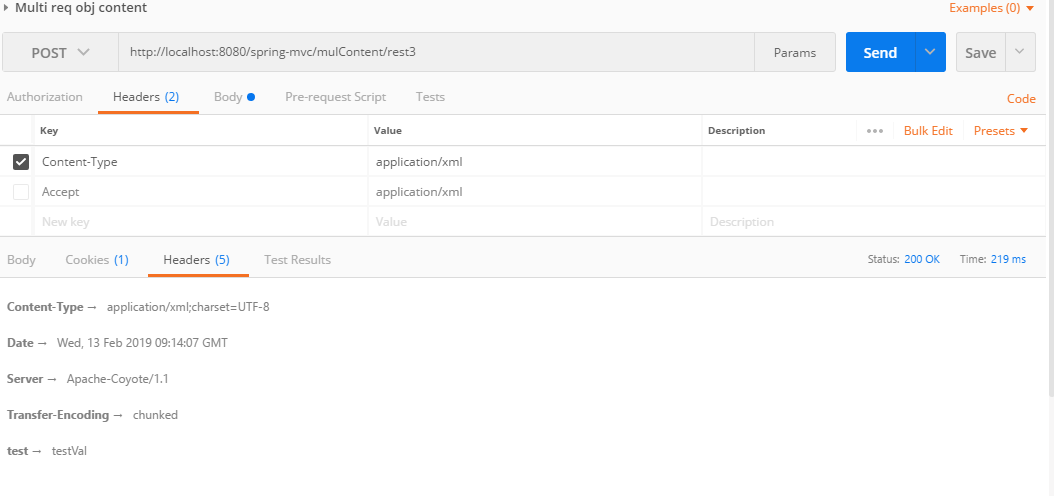
Now we can send only content-type as “application/xml”

**HttpMessageConverters**



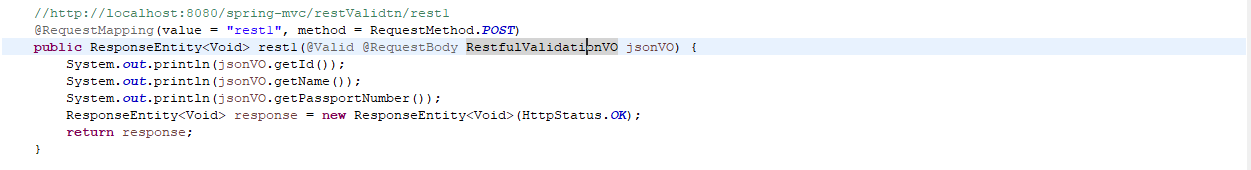
**Response Headers**



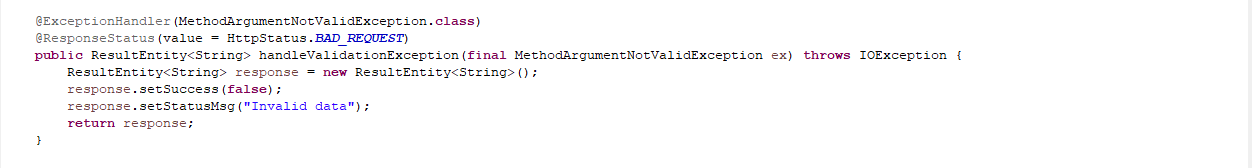


**VALIDATION**

Similar to MVC, Restful will throw ArgumentNotValidException, handle it for errors

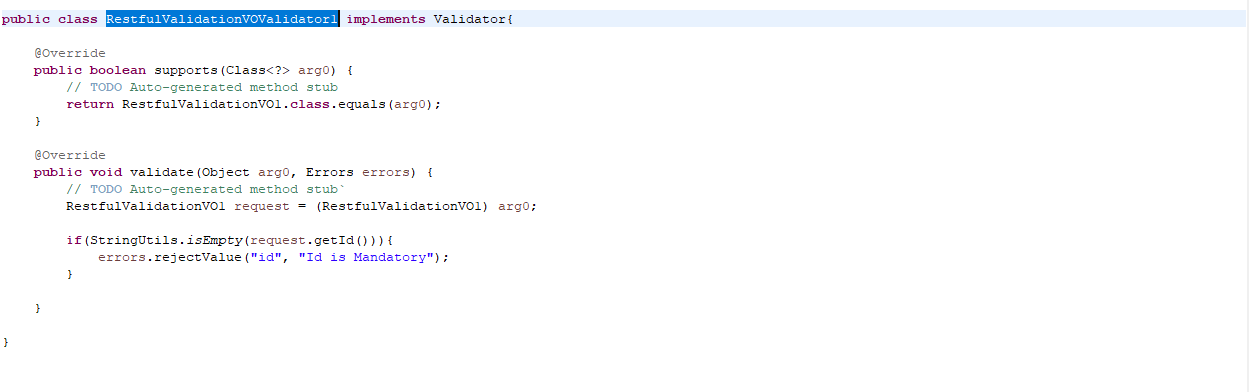






Also we can use init binder





**SPRING HATEOAS**

Spring HATEOAS provides ResourceAssembler interface to make it possible to use dedicated classes responsible for converting domain types to REST resource types. This helps us not to repeat resources and their links creation at multiple places in controller classes.

In this example we will extend ResourceAssemblerSupport (an implementation of ResourceAssembler) to understand how that works.



**JACKSON ANNOTATIONS**

**Link:** https://springframework.guru/jackson-annotations-json/

**Properties for both serialization and de-serialization**

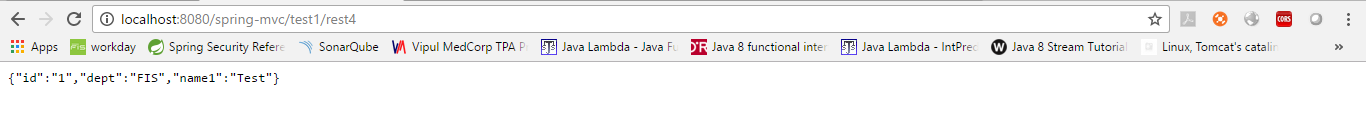
**@JsonProperty**

The @JsonProperty annotation is used to map property names with JSON keys during serialization and deserialization. By default, if you try to serialize a POJO, the generated JSON will have keys mapped to the fields of the POJO. If you want to override this behavior, you can use the @JsonProperty annotation on the fields. It takes a String attribute that specifies the name that should be mapped to the field during serialization.

You can also use this annotation during deserialization when the property names of the JSON and the field names of the Java object do not match.

Change the json property

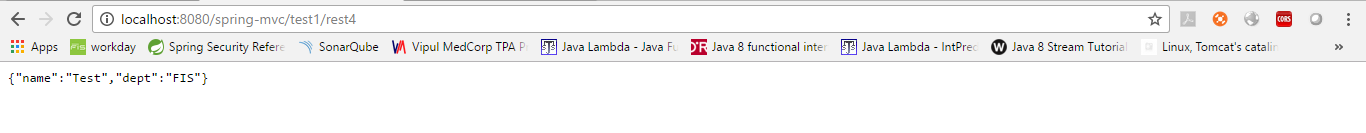




**@JsonIgnore**

The @JsonIgnore annotation marks a field of a POJO to be ignored by Jackson during serialization and deserialization. Jackson ignores the field both JSON serialization and deserialization. An example of Java class that uses the @JsonIgnore annotation is this.





**@JsonIgnoreProperties**

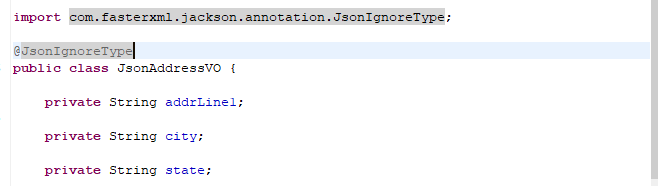
The @JsonIgnoreProperties annotation is used at the class level to ignore fields during serialization and deserialization. The properties that are declared in this annotation will not be mapped to the JSON content.

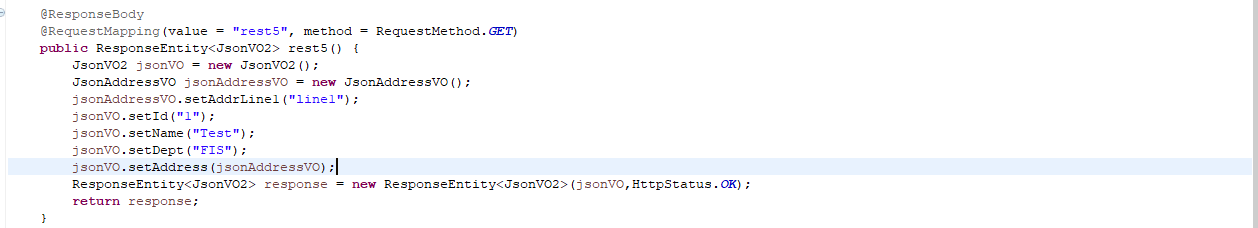


**@JsonIgnoreType**

The @JsonIgnoreType annotation is used to mark a class to be ignored during serialization and deserialization. It marks all the properties of the class to be ignored while generating JSON and reading JSON. An example of Java class that uses the @JsonIgnoreType annotation is this.







**@JsonUnwrapped**

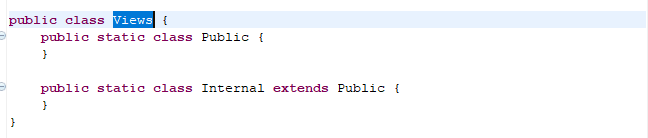
The @JsonUnwrapped annotation unwraps the values during serialization and deserialization. It helps in rendering the values of a composed class as if they belonged to the parent class. Let us consider an example of Java class that uses the @JsonUnwrapped annotation.

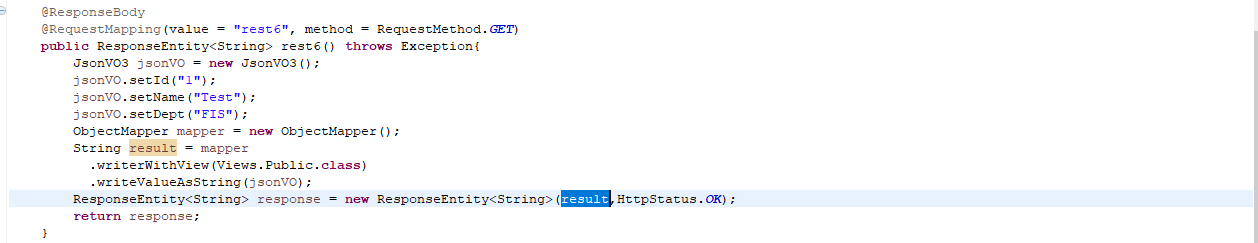


**@JsonView**

The @JsonView annotation is used to include or exclude a property dynamically during serialization and deserialization. It tells the view in which the properties are rendered. Let us consider an example Java class that uses the @JsonView annotation with Public and Internal views.







**@JsonAutoDetect**

The @JsonAutoDetect annotation is used at the class level to tell Jackson to override the visibility of the properties of a class during serialization and deserialization. You can set the visibility with the following elements:

* creatorVisibility
* fieldVisibility
* getterVisibility
* setterVisibility
* isGetterVisibility

The JsonAutoDetect class defines public static constants that are similar to Java class visibility levels. They are:

* ANY
* DEFAULT
* NON\_PRIVATE
* NONE
* PROTECTED\_AND\_PRIVATE
* PUBLIC\_ONLY

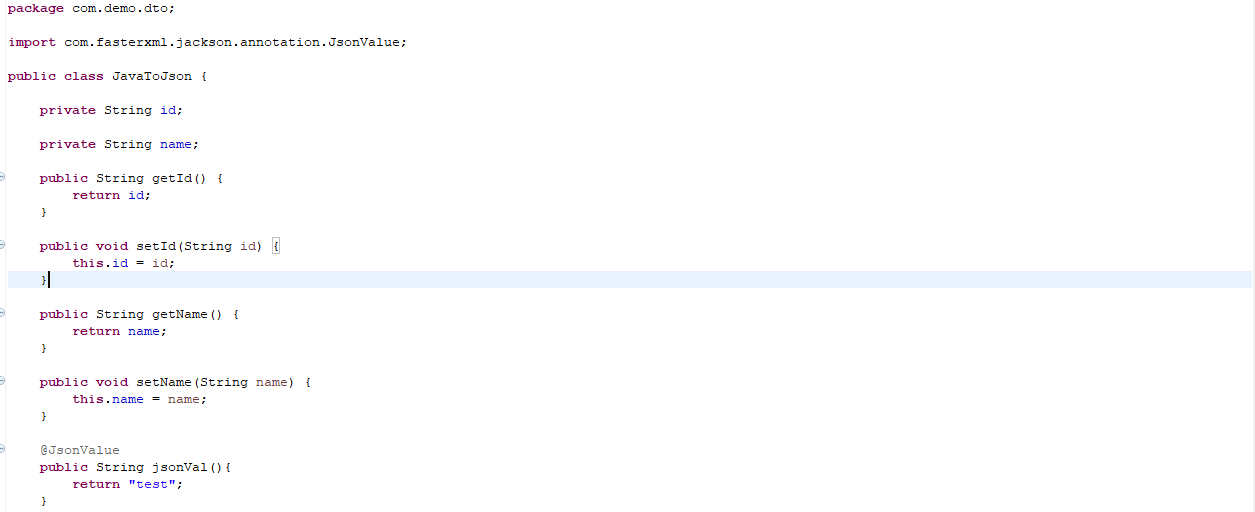
**Serialization Annotations - Java objects to JSON**

**@JsonValue**

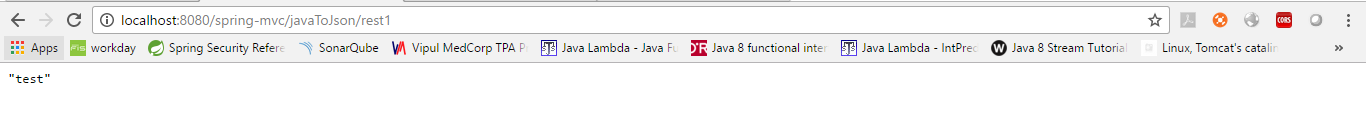
The @JsonValue annotation is used at the method level. This annotation tells Jackson to use this method to generate the JSON string from the Java object.

Note: Jackson omits any quotation marks inside the String returned by the custom serializer.

Let us consider an example Java class that uses the @JsonValue annotation.







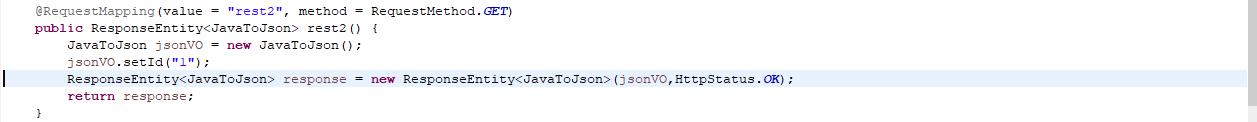
**@JsonInclude**

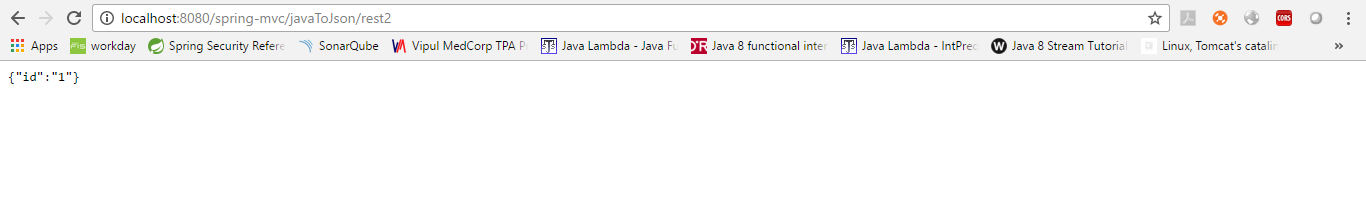
The @JsonInclude annotation is used to exclude properties or fields of a class under certain conditions. This is defined using the JsonInclude.Include enum. This enum contains constants, that determine whether or not to exclude the property. The constants are:

* ALWAYS
* NON\_DEFAULT
* NON\_EMPTY
* NON\_NULL

Let us consider an example Java class that uses the @JsonInclude annotation.



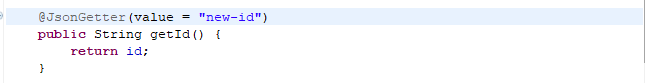




**@JsonGetter**

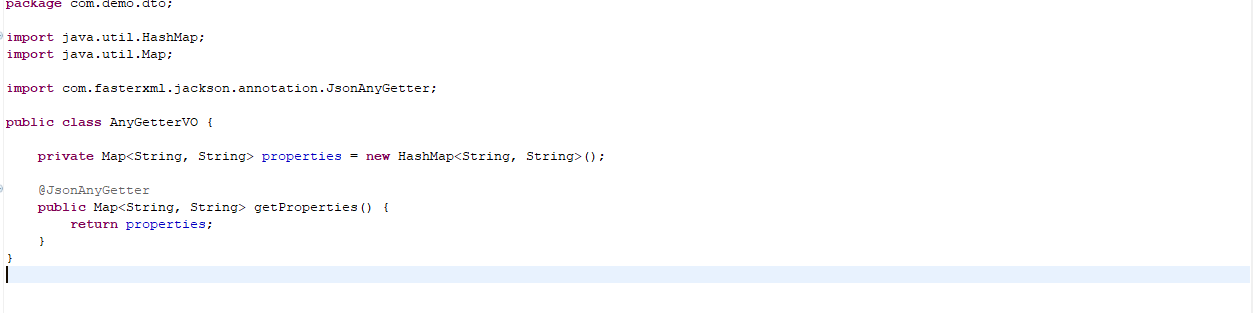
The @JsonGetter annotation is used to customize the generated JSON keys. This is accomplished with the value argument of @JsonGetter. The value passed is the name that should be used as the JSON key.

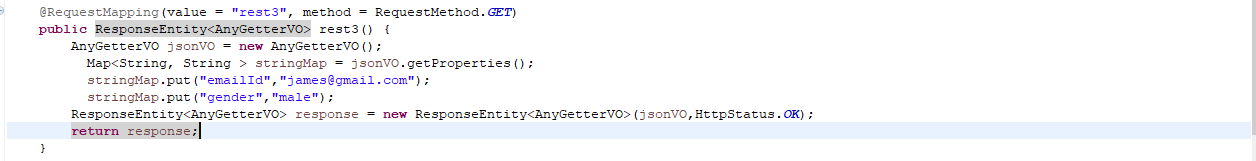
Let us consider an example Java class that uses the @JsonGetter annotation.



**@JsonAnyGetter**

The @JsonAnyGetter annotation can be used when you don’t want to declare a property or a method for every possible key in JSON. This annotation is used on the getter methods which enables you to use a Map to hold all your properties that you want to serialize.

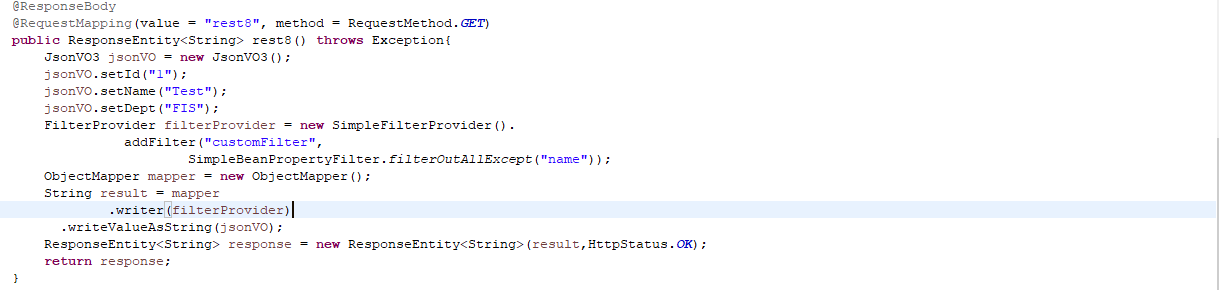




**@JsonFilter**

The @JsonFilter annotation is used to tell Jackson to use a custom defined filter to serialize the Java object. To define your filter, you need to use the FilterProvider class. This provider gets the actual filter instance to use. The filter is then configured by assigning the FilterProvider to ObjectMapper.





**@JsonFormat**

The @JsonFormat annotation is used to tell Jackson that the format in which the value for a field is serialized. It specifies the format using the JsonFormat.Shape enum.

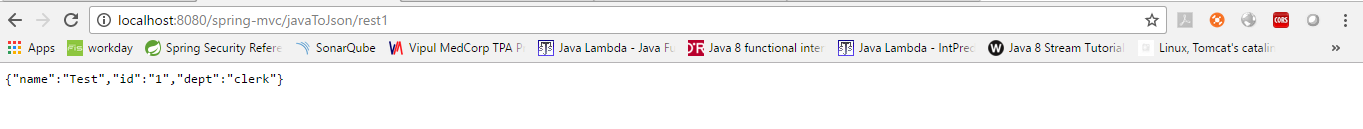


**@JsonPropertyOrder**

The @JsonPropertyOrder annotation tells Jackson to serialize the Java object to JSON, in the order specified as the arguments of the annotation. This annotation also allows partial ordering. The properties are first serialized in the order in which they are found, followed by any other properties not included in the annotation.

Let us consider an example of Java class that uses the @JsonPropertyOrder annotation.

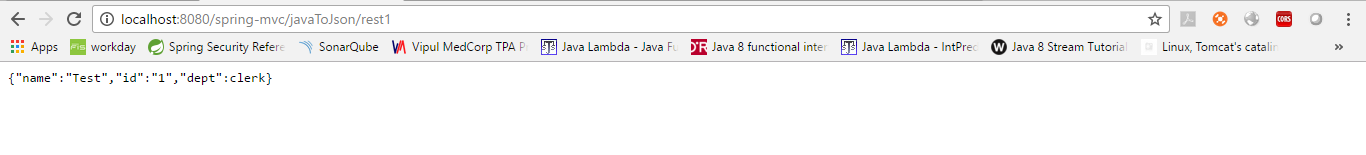




**@JsonRawValue**

The @JsonRawValue annotation is used on methods and fields. It tells Jackson to serialize the field or property as declared. For example, if you have a String field in your Java class, the JSON value that Jackson generates is enclosed within quotes (” “). But when you annotate the field with @JsonRawValue, Jackson omits the quotes.





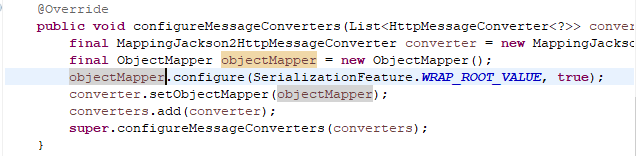
**@JsonSerialize**

The @JsonSerialize annotation is used tell Jackson to use the declared custom serializer during the serialization of the field, which is marked with this annotation. Let us consider a POJO that uses the @JsonSerialize annotation.

**@JsonRootName**

The @JsonRootName annotation can be used to tell Jackson to wrap the object to be serialized with a top-level element. You can pass the name as a parameter to the @JsonRootName annotation. Let us consider that you want to wrap your serialized Java object with the key user.

Enable wrap root name







**Deserialization Annotations**

**@JsonSetter**

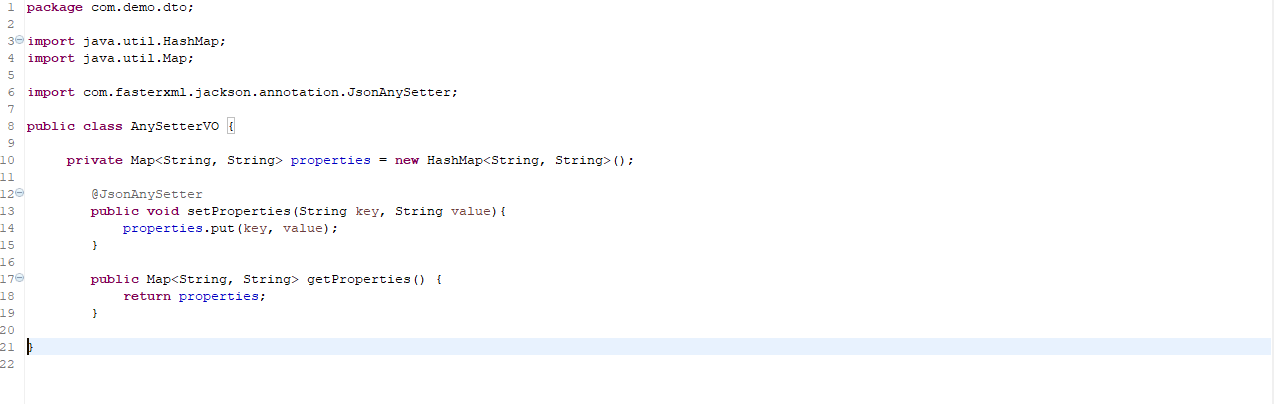
The @JsonSetter annotation tells Jackson to deserialize the JSON into Java object using the name given in the setter method. Use this annotation when your JSON property names are different to the fields of the Java object class, and you want to map them.

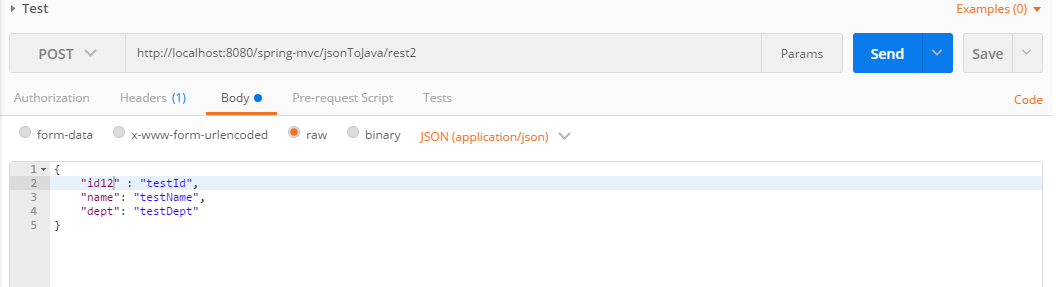




**@JsonAnySetter**

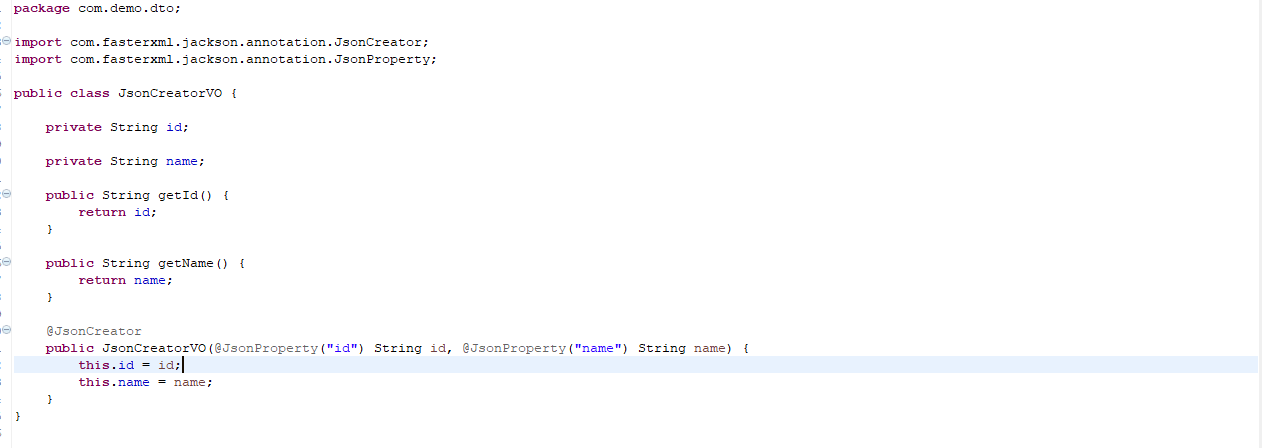
The @JsonAnySetter annotation is used on setter methods of a Map field. Sometimes you may find some JSON values that cannot be mapped to the fields in the Java object class. In such a case, the @JsonAnySetter captures the data and stores them in a Map.

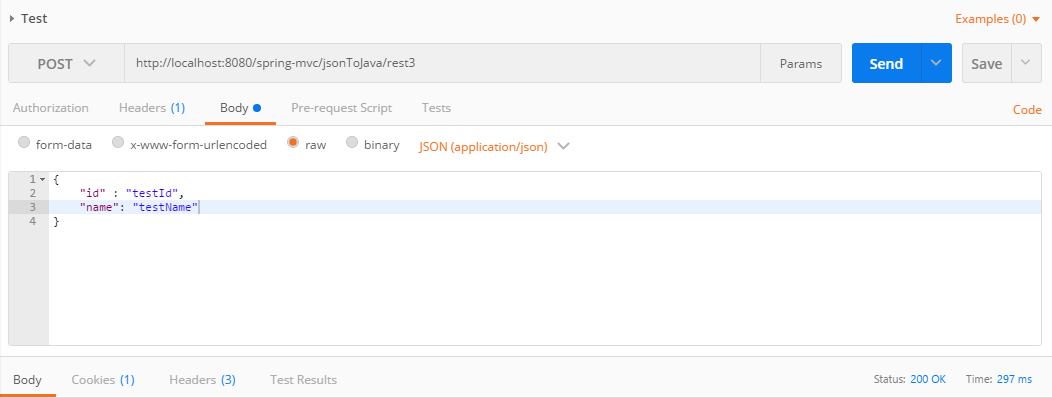




**@JsonCreator**

The @JsonCreator annotation tells Jackson that the JSON properties can be mapped to the fields of a constructor of the POJO. This is helpful when the JSON properties do not match with the names of the Java object field names. The @JsonCreator annotation can be used where @JsonSetter cannot be used. For example, immutable objects which need their initial values to be injected through constructors.

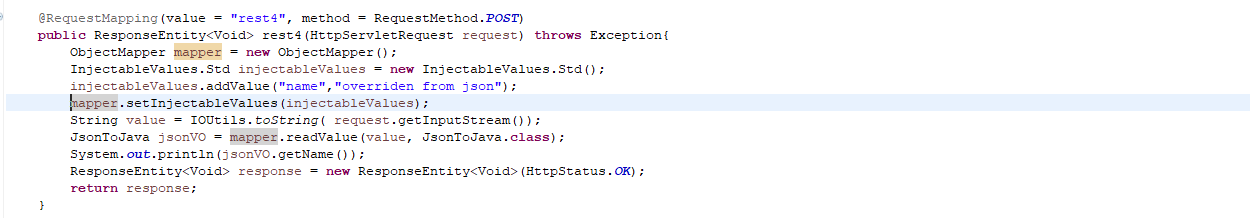




**@JacksonInject**

The @JacksonInject annotation is used to tell Jackson that particular values of the deserialized object will be injected and not read from the JSON string.





**@JsonDeserialize**

The @JsonDeserialize annotation tells Jackson to use a custom deserializer while deserializing the JSON to Java object. To do so, you need to annotate the field to which you need to apply the custom deserializer.

**RESTTEMPLATE**

**Configuration**

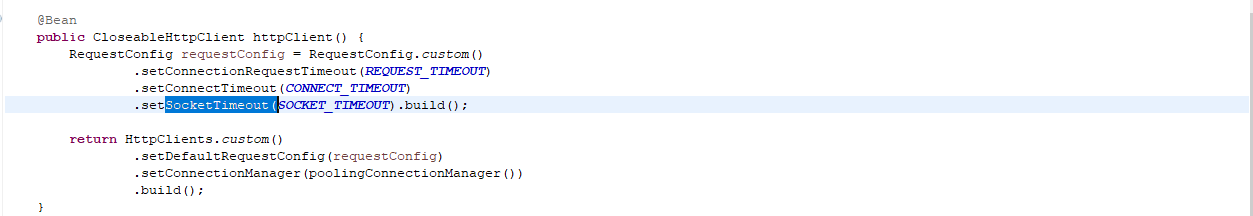
**HttpClient Connection Management**

The PoolingHttpClientConnectionManager will create and manage a pool of connections for each route or target host we use. The default size of the pool of concurrent connections that can be open by the manager is 2 for each route or target host, and 20 for total open connections. First – let’s take a look at how to set up this connection manager on a simple HttpClient:



* setMaxTotal(int max): Set the maximum number of total open connections.
* setDefaultMaxPerRoute(int max): Set the maximum number of concurrent connections per route, which is 2 by default.
* setMaxPerRoute(int max): Set the total number of concurrent connections to a specific route, which is 2 by default.

**HttpClient**



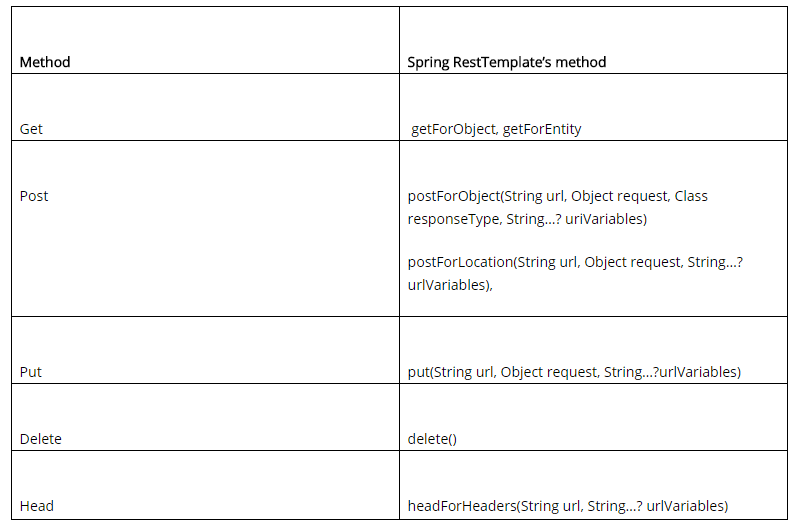
* the Connection Timeout (http.connection.timeout) – the time to establish the connection with the remote host
* the Socket Timeout (http.socket.timeout) – the time waiting for data – after the connection was established; maximum time of inactivity between two data packets
* the Connection Request Timeout (http.connection-manager.timeout) – the time to wait for a connection from the connection manager/pool

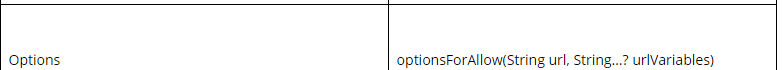
**Intro**

In Spring REST client, The RestTemplate is the core class for client-side access to Spring RESTful web services. It provides the methods to communicate by using these HTTP methods with URI template, URI param, request object and response type as arguments.

**RestTemplate Methods**

RestTemplate provides higher level methods that correspond to each of the six main HTTP methods that make invoking many RESTful services. In the following list has methods provided by Spring RestTemplate for each http methods.

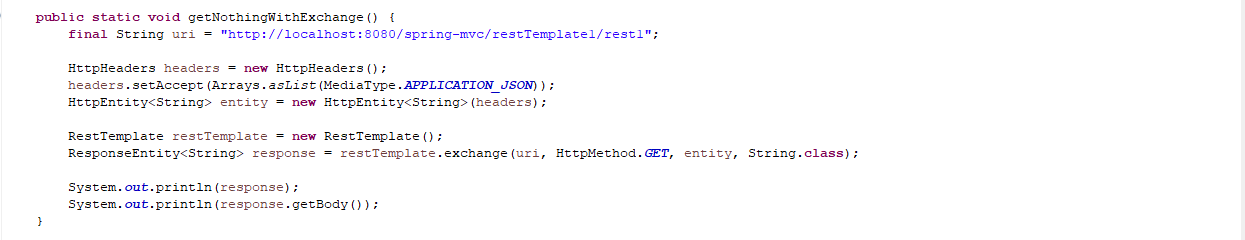




As per as name of these methods of RestTemplate class indicate which HTTP method they implement internally and second part of the method name indicates what is return by these methods. For example, getForObject() will perform a HTTP GET action on the server, and convert the HTTP response into an object of given type and return it to the client. And other method postForLocation() will do a HTTP POST action on the server, converting the given object into a HTTP request, and returns the response HTTP Location header where the newly created object can be found.

* Use HttpHeaders to set the Request Headers.
* Use HttpEntity to wrap the request object.
* Provide the URL, HttpMethod, and Return type for Exchange() method.

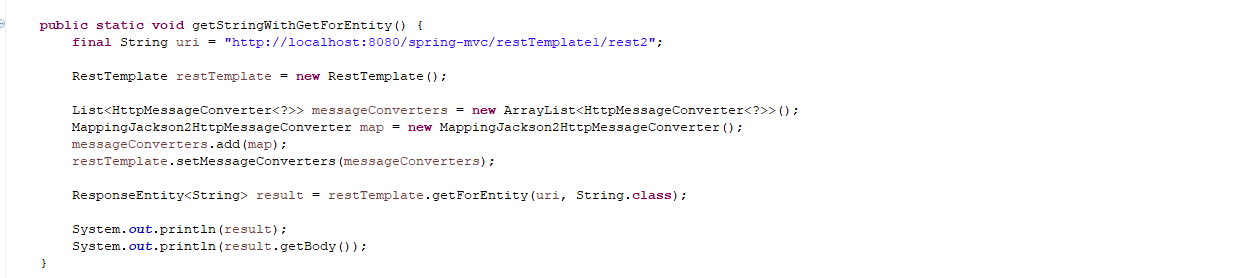
Using exchange



getForObject

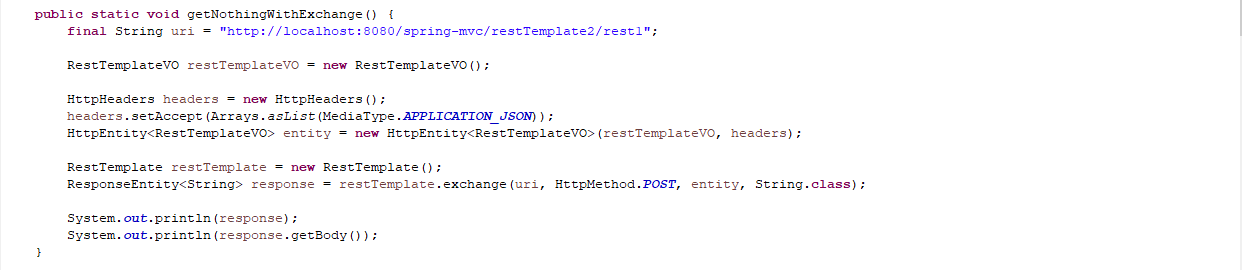


getForEntity

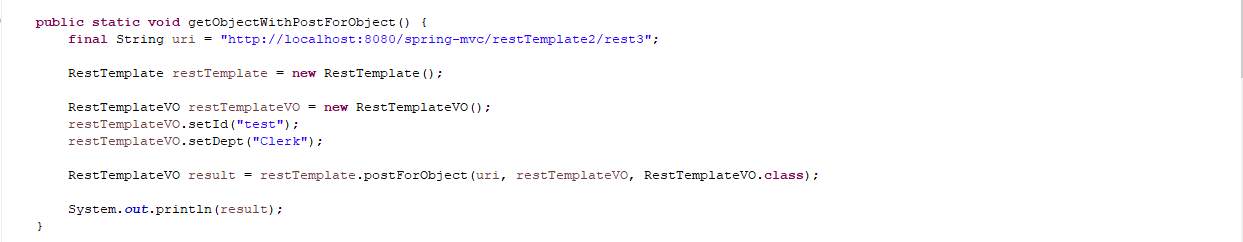


**POST**

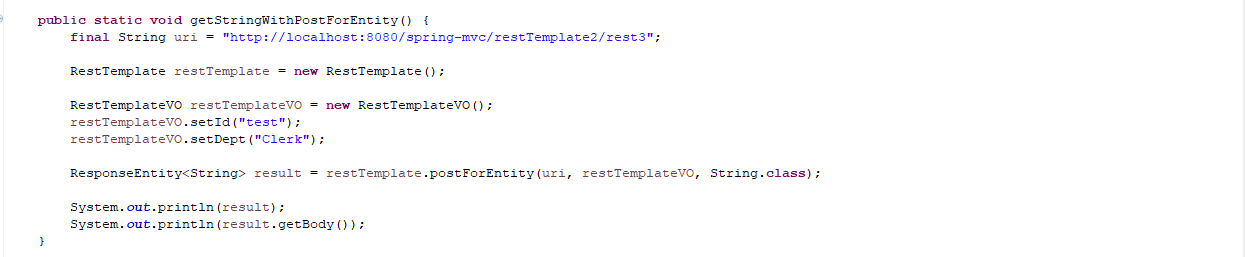
With Exchange



postForObject

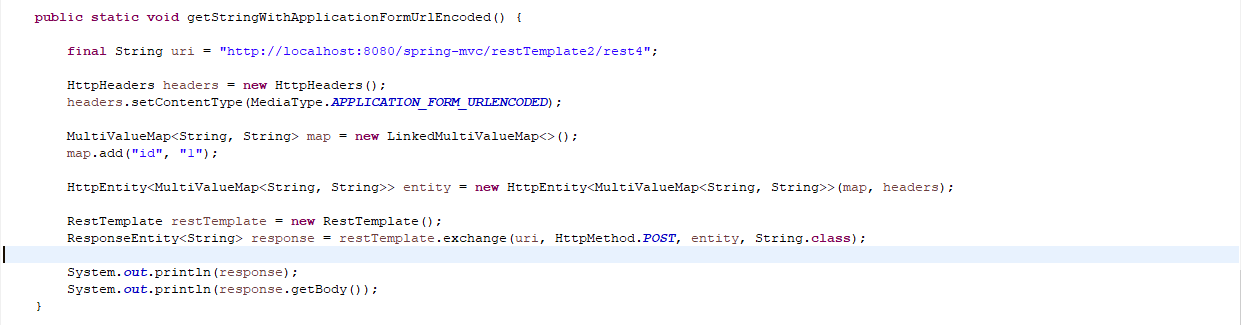


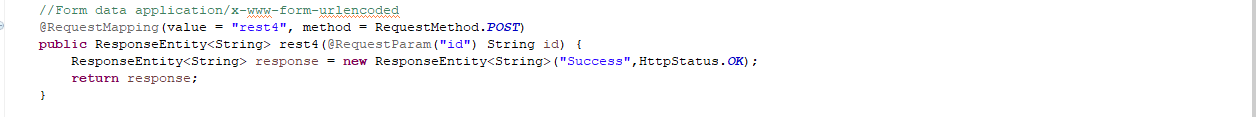
postForEntity



**FORM DATA**

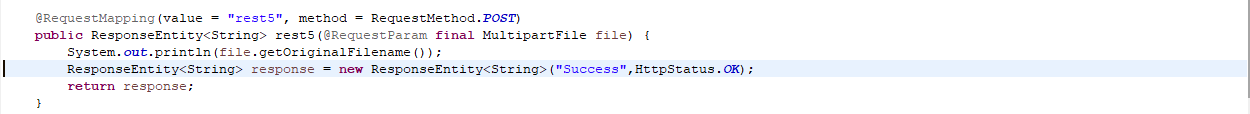
**application/x-www-form-urlencoded**





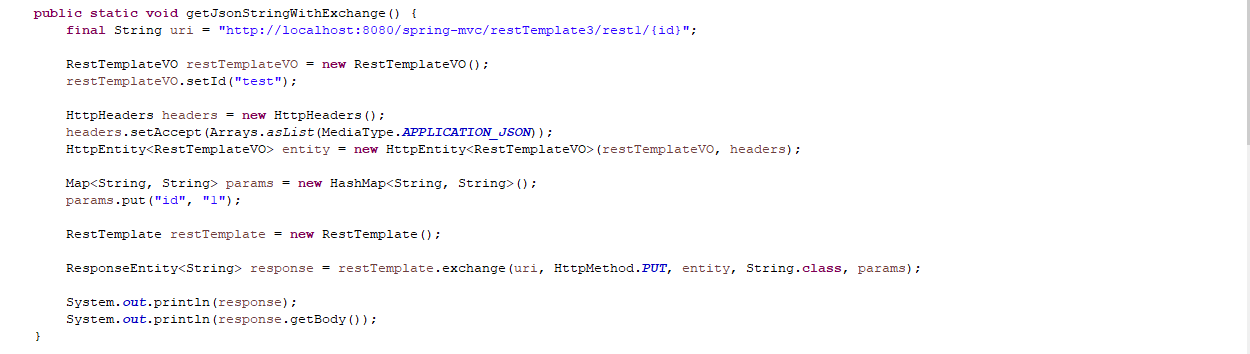
**multipart/form-data**



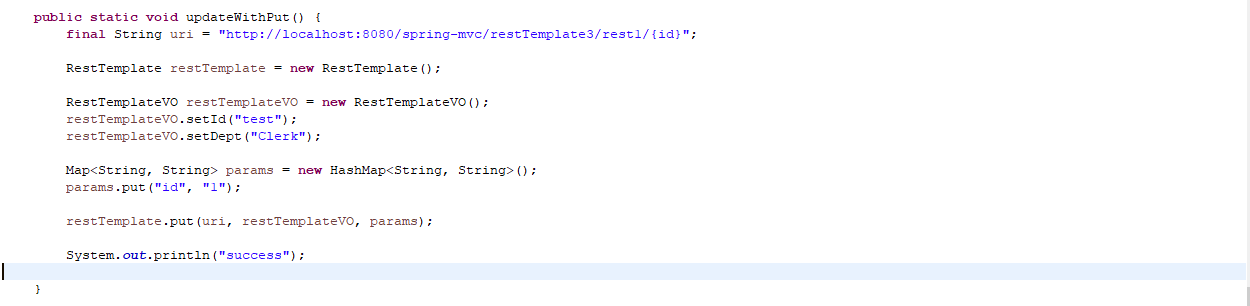


**PUT**

Exchange

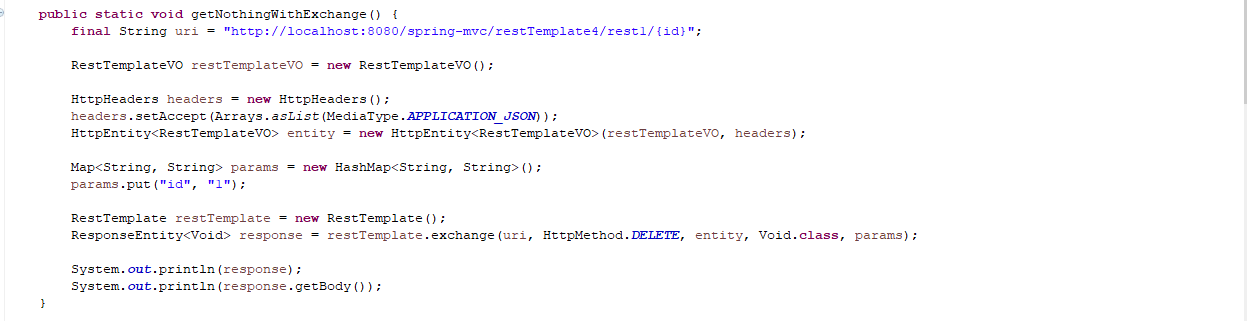


Put



**DELETE**

With Exchange



With delete method

