| **Spring controller injected parameters**  **Supported method arguments types** | | |
| --- | --- | --- |
| **Supported**  **arguments** | **Use/description** | **Packages** |
| ServletRequest /  HttpServletRequest | Injects the servlet request/response. | javax.servlet.http.\* |
| ServletResponse /  HttpServletResponse |
| HttpSession | Injects the HTTP session bound to the servlet request. If null, Spring creates a new one.  synchronizeOnSession must be set on an AbstractController or in RequestMappingHandlerAdapter if sessions should be shared  concurrently across multiple requests. |
| WebRequest / NativeWebRequest | Injects a wrapper for access to request parameters and request/session attributes only. | org.springframework.web.context.request.\* |
| Locale | Injects the local e of the request using the configured LocaleResolver. | java.util.\* |
| InputStream / Reader | Provides a direct access to the request/response payload. | java.io.\* |
| OutputStream / Writer |
| HttpMethod | Injects the current method of the request. | org.springframework.http.\* |
| Principal | Using the Spring security context, it injects the authenticated account. | java.security.\* |
| HttpEntity<?> | Spring converts and injects the inbound request to a custom type using HttpMessageConverter. It also provides access to the request headers. | org.springframework.http.\* |
| Map | Instantiates for us a BindingAwareModelMap to be used in the view. | java.util.\* |
| Model | org.springframework.ui.\* |
| ModelMap |
| RedirectAttributes | Injects and repopulates a map of attributes and flash attributes maintained over request redirection | org.springframework.web.servlet.mvc.support.\* |
| Errors | Injects the validation results of the argument located just before in the argument list. | org.springframework.validation.\* |
| BindingResult |
| SessionStatus | Allows tagging with setComplete(Boolean), the completion of a session. This method clears the session attributes defined at the type level with @SessionAttributes. | org.springframework.web.bind.support.\* |
| UriComponentsBuilder | Injects a Spring URL builder UriComponentsBuilder. | org.springframework.web.util.\* |

#### Supported annotations for method arguments

| **Supported annotation arguments** | **Use/description** | **Package** |
| --- | --- | --- |
| @PathVariable | Injects an URI Template variable into an argument. | org.springframework.web.bind.annotation.\* |
| @MatrixVariable | Injects name-value pairs located in URI path segments into an argument. |
| @RequestParam | Injects a specific request parameter into an argument. |
| @RequestHeader | Injects a specific request HTTP Header into an argument. |
| @RequestBody | Allows direct access to the request payload injecting it into an argument. |
| @RequestPart | Injects the content of a specific part (meta-data, file-data…) of a multipart/form-data encoded request into an argument of the matching type (MetaData, MultipartFile…) |
| @ModelAttribute | Populates automatically an attribute of the Model using the URI template.  This binding is operated before the method handler processing. |

| **Supported return Types** | **Use/description** | **Packages** |
| --- | --- | --- |
| Model | Spring MVC creates an implementation of the Model Interface for the handler method.  The Model objects are populated manually within the handler- method or with @ModelAttribute.  The view to render needs to be mapped to the request with RequestToViewNameTranslator. | org.springframework.ui.\* |
| ModelAndView | A wrapper object for the Model with a View and a view name. If a view name is provided, Spring MVC will attempt to resolve the associated View. Otherwise, the embedded View is rendered.  The Model objects are populated manually within the method or with @ModelAttribute. |
| Map | Allows a custom Model implementation.  The view to render needs to be mapped to the request with RequestToViewNameTranslator. | java.util.\* |
| View | Allows the rendering of a custom View object.  Spring MVC creates an implementation of the Model interface for the handler method.  The objects of the model are populated manually within the method or with the help of @ModelAttribute. | org.springframework.web.servlet.\* |
| String | If a @ResponseBody annotation is not specified on the handler method, the returned String is processed as a View name (View identifier). | java.lang.\* |
| HttpEntity<?> / ResponseEntity<?> | Two wrapper objects to easily manage the response headers and converted-by-Spring body (with HttpMessageConverters). | org.springframework.http.\* |
| HttpHeaders | Provides a wrapper object for HEAD responses. | org.springframework.http .\* |
| Callable<?> | Can produce asynchronously a typed object when the Thread is controlled by Spring MVC. | java.util.concurrent.\* |
| DeferredResult<?> | Can produce asynchronously a Typed object when the Thread is not controlled by Spring MVC. | org.springframework.web.context.request.async.\* |
| ListenableFuture<?> | org.springframework.util.concurrent.\* |
| void | When the view is externally resolved with RequestToViewNameTranslator or when the method prints directly in the response. |  |

WebContentInterceptor:  This superclass allows us to globally control sessions and to manage caching options. . As an interceptor, it must be registered declaratively. This is the reason why we have added two <mvc:interceptors> entries in our context files.

setRequireSession(false); This allows defining whether or not a session should be required when handling a request. If there is no session bound to the request (if the session has expired for example), the controller will throw a SessionRequiredException method. In such cases, it is good to have a global ExceptionHandler defined.

setCacheSeconds(0);

Set to zero, it adds the extra headers in the response such as Pragma, Expires, Cache-control, and so on.

setSupportedMethods("GET","POST", "OPTIONS", "HEAD");

<bean class="org.sfw...annotation.RequestMappingHandlerAdapter">

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

</bean>

synchronizeOnSession: When set it to true, the session object is serialized and access to it is made in a synchronized block. This allows concurrent access to identical sessions and avoids issues that sometimes occur when using multiple browser windows or tabs.

with WebApplicationObjectSupport, WebContentGenerator provides:

* Access to ServletContext out of the request or response object through getServletContext().
* Access to the temporary directory for the current web application, as provided by the servlet container through getTempDir().
* Access to the WebApplicationContext through getWebApplicationContext().
* Also, a couple of tools to set and initialize the ServletContext and the WebApplicationContext, even if these tools are initially intended for use within the Framework itself.

**@EnableWebMvc**. With this annotation, we are telling Spring MVC to configure the DefaultAnnotationHandlerMapping, AnnotationMethodHandlerAdapter and ExceptionHandlerExceptionResolver beans. These beans are required for Spring MVC to dispatch requests to the controllers. n. It also enables support for various convenient annotations such as @NumberFormat, @DateTimeFormat to format the form bean's fields during form binding, and similarly the @Valid annotation to validate the controller method's parameters. It even supports Java objects being converted to/from XML or JSON via the @RequestBody and @ResponseBody annotation in the @RequestMapping or @ExceptionHandler methods during form binding. So @EnableWebMvc annotation is needed to enable annotations such as @controller and @RequestMapping and so on.

RequestMappingHandlerAdapter :It's used with RequestMappingHandlerMapping class, which **executes methods annotated with @RequestMapping**.

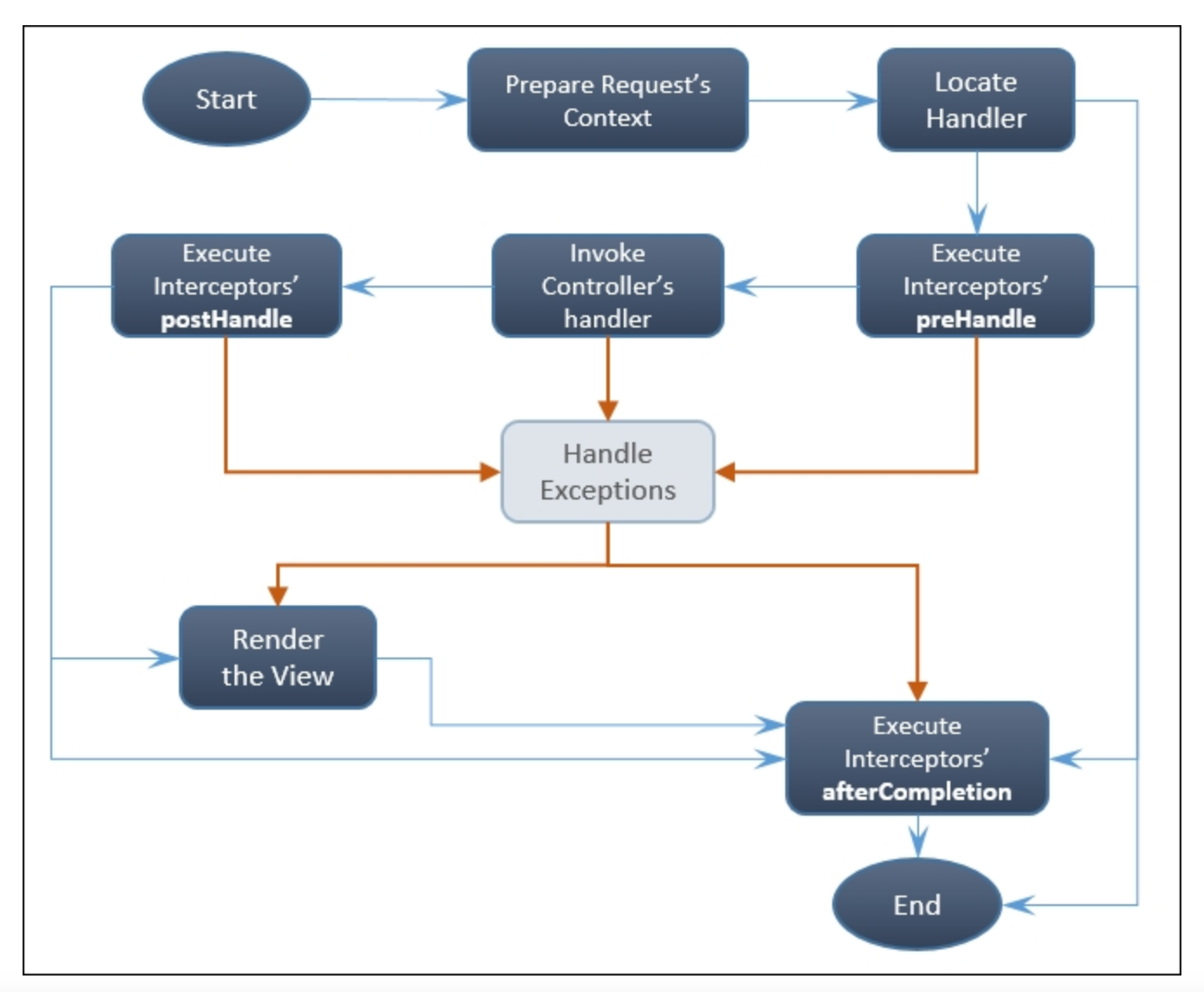
Localization: specific to particular locale

Internationalization: For more than 1 locale

**Interceptor lifecycle:**

### REQUEST LIFECYCLE

Throughout the interceptor(s), each request is processed according to the following lifecycle:

* Prepare the request's context
* Locate the Controller's handler
* Execute interceptor's preHandle methods
* Invoke the Controller's handler
* Execute interceptor's postHandle methods
* Handle the Exceptions
* Process the View
* Execute interceptor's afterCompletion methods
* 
* The controller handler is invoked, unless one of the interceptors' preHandle methods throws an exception.
* An interceptor's postHandle method is called when the controller's handler finishes without throwing an exception and if no preceding postHandler method has thrown an exception.
* An interceptor's afterCompletion is always called, unless a preceding afterCompletion throws an exception.

**Spring taglib:**

<%@taglib prefix="form" uri="http://www.springframework.org/tags/form" %>

<%@taglib prefix="spring" uri="http://www.springframework.org/tags" %>

If we have data in the model, we can display it on the view

Model.addAttribute(“user”, user);

${user.name}

But if data is on html, we want it to be mapped to a bean.

 Spring tag library tags help us to bind the HTML tag element's values to a form backing bean in the Model. Later, the Controller can retrieve the form backing bean from the Model using the @ModelAttribute (org.springframework.web.bind.annotation.ModelAttribute) annotation.

The Spring tag library provides some special <form> and <input> tags, which are more or less similar to HTML form and input tags, but have some special attributes to bind form elements' data with the form backed bean

In the Spring <form:form> tag, one of the important attributes is modelAttribute.

The path attribute just indicates the field name that is relative to the form backing bean.

 redirect:/market/products

WebDataBinder extracts the data out of the HttpServletRequest object and converts it to a proper data format, loads it into a form backing bean, and validates it. To customize the behavior of data binding, we can initialize and configure the WebDataBinder object in our Controller. The @InitBinder (org.springframework.web.bind.annotation.InitBinder) annotation helps us to do that. The @InitBinder annotation designates a method to initialize WebDataBinder. , WebDataBinder internally uses many PropertyEditor (java.beans.PropertyEditor) implementations to convert the HTTP request parameters to the target field of the form backing bean. We can even register custom PropertyEditor objects with WebDataBinder to convert more complex data types.

@InitBinder

public void initialiseBinder (WebDataBinder binder) {

DateFormat dateFormat = new SimpleDateFormat("MMM d, YYYY");

CustomDateEditor orderDateEditor = new CustomDateEditor(dateFormat, true);

binder.registerCustomEditor(Date.class, orderDateEditor);

}

we have a label tag with the hardcoded text value as Product id:

<label class="control-label col-lg-2 col-lg-2" for="productId">**Product Id**</label>

externalize locale-sensitive text messages from a web page to a property file.

    <label class="control-label col-lg-2 col-lg-2"

      for="productId"> <spring:message

      code="addProduct.form.productId.label"/> </label>

Create a file called messages.properties under /src/main/resources in your project and add the following line to it:

      addProduct.form.productId.label = New Product ID

Now open our web application context configuration file WebApplicationContextConfig.java and add the following bean definition to it:

      @Bean

      public MessageSource messageSource() {

       ResourceBundleMessageSource resource = new

 ResourceBundleMessageSource();

       resource.setBasename("messages");

       return resource;

      }

One important property you need to notice here is the basename property; we assigned the value messages for that property.

Spring MVC has a special a tag called <spring:message> to externalize texts from JSP files. In order to use this tag, we need to add a reference to a Spring tag library; his code attribute is a kind of key; at runtime Spring will try to read the corresponding value for the given key (code) from a message source property file.

n Spring MVC, the field error messages are generated by validators associated with the controller, and you can use the **<form:errors />** tag to render those field error messages in an default HTML “**span**” tag.

And then, you can use the **<form:errors />** to render the error messages that are associated with the “**username**” field.

<form:errors path="userName" cssClass="error" />

It will renders and enclose the error messages with a default “**span**” element, which contains a CSS class of “**error**“.

<span id="username.errors" class="error">username is required!</span>

1. path=”\*” – display all error messages associated with any fields.
2. path=”username” – display error messages associated with the “username” field only.
3. For some reasons, like CSS formatting purpose, you may need to enclose the error messages with different element instead of the default “**span**” tag. To do this, just specify the prefer element inside the “**element**” attribute :
4. <form:errors path="userName" cssClass="error" element="div" />
5. Copy
6. Now, it renders and enclose the error messages with a “**div**” element, which contains a CSS class of “**error**“.
7. <div id="username.errors" class="error">username is required!</div>

Can move error messaes to property file

@Size( max=40, min=6)

String name;

Size.user.name=**{0}** must be between **{2}** and **{1}** in length

{0}: name

{1}: 40; arranged alphabetically

**FlashAttribute**:

Making a post request and resubmitting a form/ refresh/ F5, leads to another post request being made. If we do redirect instead, takes more time, it adds one more problem of retrieving request parameters and attributes from the initial POST request. Flash attributes can help in such cases. Flash attributes provide a way for us to store information that is intended to be used in another request. Flash attributes are saved temporarily in a session to be available for an immediate request after redirection.

In order to use Flash attributes in your Spring MVC application, just add the RedirectAttributes redirectAttributes parameter to your Spring Controller's method as follows:

@RequestMapping

public String welcome(Model model, **RedirectAttributes redirectAttributes**) {

model.addAttribute("greeting", "Welcome to Web Store!");

model.addAttribute("tagline", "The one and only amazing web store");

**redirectAttributes.addFlashAttribute("greeting", "Welcome to Web Store!");**

**redirectAttributes.addFlashAttribute("tagline", "The one and only amazing web store");**

return "redirect:/welcome/greeting";

}

Note how we used redirectAttributes parameter on method welcome to map flash attributes. Also we used addFlashAttribute method to set new parameters to flash attribute.

That is

If we have a method

@PostMapping("/user")

**public** String saveUser(@Valid @ModelAttribute User user, BindingResult result, Model m) {

**if**(result.hasErrors()) {

m.addAttribute("errors",result.getAllErrors());

System.***out***.println(result.getAllErrors());

}

System.***out***.println(user);

**return** "display";

}

On display.jsp we want to print user object

If we do refresh on display.jsp, it will again fire a post request to user

If instead of forwarding request, we redirect the request to display, we won’t be able to access user object. SO we use flash attribute and then do redirect. The attributes would be there only for first request.

**Serving static resources:**

To serve static content like images, override addResourceHandlers

**public** **void** addResourceHandlers(ResourceHandlerRegistry registry) {

//if a request comes for img, serve content from images folder

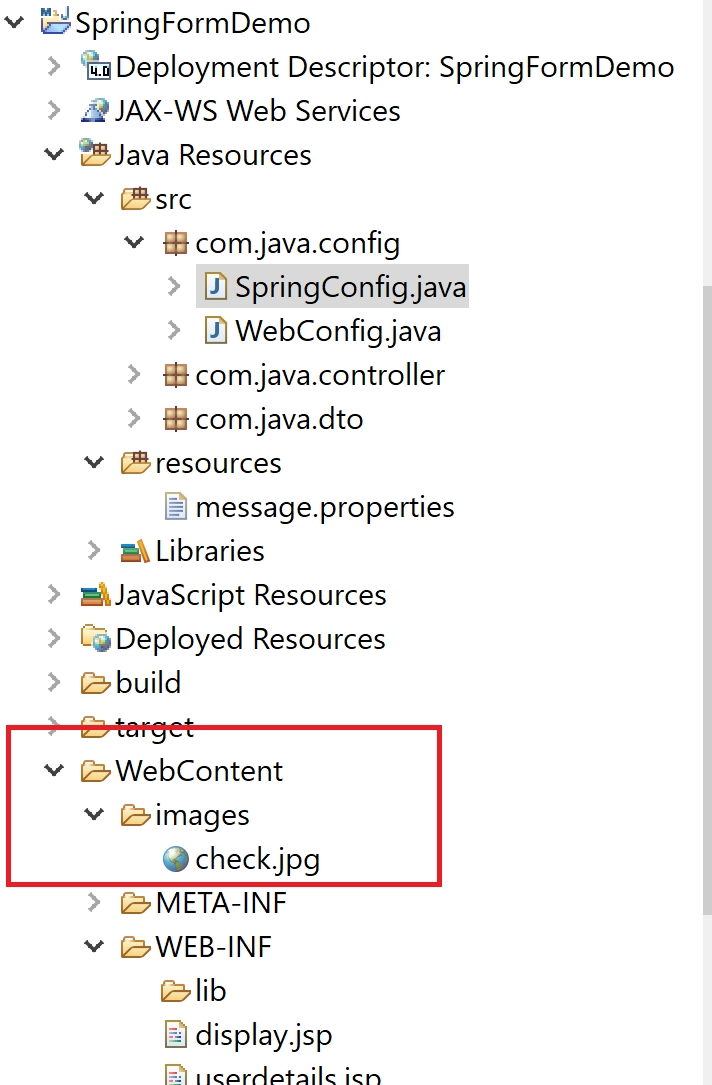
//images folder should be present as a root folder in your war file.

registry.addResourceHandler("/img/\*\*").addResourceLocations("/images/")

.setCacheControl(CacheControl.*maxAge*(6, TimeUnit.***DAYS***));

} The addResourceLocations method from ResourceHandlerRegistry defines the base directory location of the static resources that you want to serve.

For a non-maven project folder structure:



You can place it inside src/main/webapp/ folder in maven

Any type of static files such as PDFs, Word documents, Excel sheets can be served like this.

Can use this url as part of jsps:



Can also use it as part of expression language

<img src=<c:url value=”/img/{product.image}”></c:url>”/>

**Uploading file to server: Multipart request:**

Add a bean definition in our web application context configuration file (WebApplicationContextConfig.java) for CommonsMultipartResolver as follows:

      @Bean

     public CommonsMultipartResolver multipartResolver() {

     CommonsMultipartResolver resolver=new

      CommonsMultipartResolver();

     resolver.setDefaultEncoding("utf-8");

     return resolver;

     }

1. In pom: A **Select Dependency** window will appear; in **Group Id** enter **commons-fileupload**, in **Artifact Id** enter **commons-fileupload**, in **Version** enter **1.2.2**, select **Scope** as **compile**, then click on the **OK** button. Similarly, add one more dependency: org.apache.commons as **Group Id**, commons-io as **Artifact Id**, 1.3.2 as **Version**, and **Scope** as **compile**, then click on the **OK** button and save pom.xml.
2. Open our product's domain class (product.java) and add a reference to org.springframework.web.multipart.MultipartFile with corresponding setters and getters as follows (don't forget to add getters and setters for this field):
3. private MultipartFile productImage;

Open addProduct.jsp,   **<form:input id="productImage" path="productImage"**

**type="file" class="form:input-large" />**

Now set the enctype attribute to multipart/form-data in the form tag as follows and save addProduct.jsp:

      <form:form modelAttribute="newProduct" class="form-

      horizontal" **enctype="multipart/form-data"**>

Open our ProductController.java

public String processAddNewProductForm(

      @ModelAttribute("newProduct") Product newProduct,

      BindingResult result, **HttpServletRequest request**) {

  MultipartFile productImage = newProduct.getProductImage();

     String rootDirectory =

      request.getSession().getServletContext().getRealPath("/");

     if (productImage!=null && !productImage.isEmpty()) {

     try {

     productImage.transferTo(new

      File(rootDirectory+"resources\\images"+

      newProduct.getProductId() + ".png"));

     } catch (Exception e) {

     throw new RuntimeException("Product Image saving

      failed", e);

      }

     }

Spring's CommonsMultipartResolver (org.springframework.web.multipart.commons.CommonsMultipartResolver) class is the thing that determines whether the given request contains multipart content and parses the given HTTP request into multipart files and parameters. And, through the setMaxUploadSize property, we set a maximum of 10,240,000 bytes as the allowed file size to be uploaded:

we added the enctype attribute to the <form:form> tag and set its value as multipart/form-data. The enctype attribute indicates how the form data should be encoded when submitting it to the server.

We wanted to save the image file in the server under the resources/images directory, as this directory structure will be available directly under the root directory of our web application at runtime. So, in order to get the root directory of our web application, we need HttpServletRequest. See the following code snippet:

String rootDirectory = request.getSession().getServletContext().getRealPath("/");

One such pre-built interceptor is **LocaleChangeInterceptor**, which allows us to change the current locale on every request and configures LocaleResolver to support internationalization.

Bydefault, it uses a AcceptHeaderLocaleResolver which intercepts every request and gets locale information from Accept header

If we want to retrieve this locale information from request parameter, we use LocaleChangeInterceptor. Bydefault using AcceptHeaderLocaleResolver.

@Bean

**public** LocaleChangeInterceptor localeChangeInterceptor() {

LocaleChangeInterceptor ic= **new** LocaleChangeInterceptor();

ic.setParamName("lang");

**return** ic;

}

@Override

**public** **void** addInterceptors(InterceptorRegistry registry) {

registry.addInterceptor(localeChangeInterceptor());

}

@Bean

**public** LocaleResolver localeResolver() {

SessionLocaleResolver rc= **new** SessionLocaleResolver();

**return** rc;

}