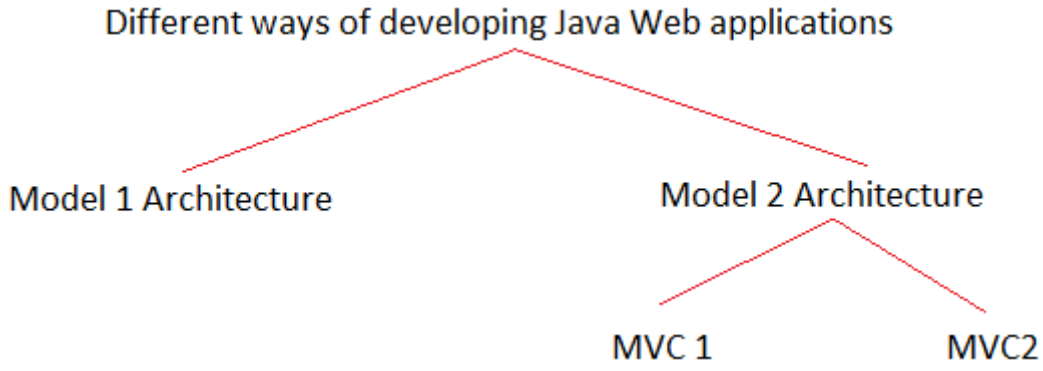
Prepared By - Nirmala Kumar Sahu

INDEX

Spring Boot MVC ----------

Spring Boot MVC

Introduction



Model 1 Architecture:

- We take either Servlet components or JSP components as the main components of the web application i.e., if Servlet components are used in web applications, then the JSP components will not be used and vice-versa.
- Here each main web components contains multiple logics that means there is no clean separation between logic. So not suitable for large scale applications.

Model 2 Architecture (MVC 1 & MVC 2):

- Here we take support of multiple technologies in multiple layers to develop the logics.
- Suitable for medium scale, large scale applications.

- M for Model Layer, represents data.
  - Generally, talks about Service layer + Persistence layer.
  - It is like Accounts officer.
  - Takes care of the calculations and analyzations of the application.

- V for View Layer, represents presentation logic.
  - It is like beautician

- C for Controller Layer, represents Monitoring logic/ Navigation logic.
  - It is like supervisor.

- MVC 2 architecture is more industry standard.
- All web frameworks like Struts, JSF, Spring MVC, Spring Boot MVC and etc. are given based in MVC 2 architecture (MVC = MVC2).
- MVC 3, MVC 4, MVC 5, MVC 6 are no way related Java architecture there are the versions of asp.net MVC.

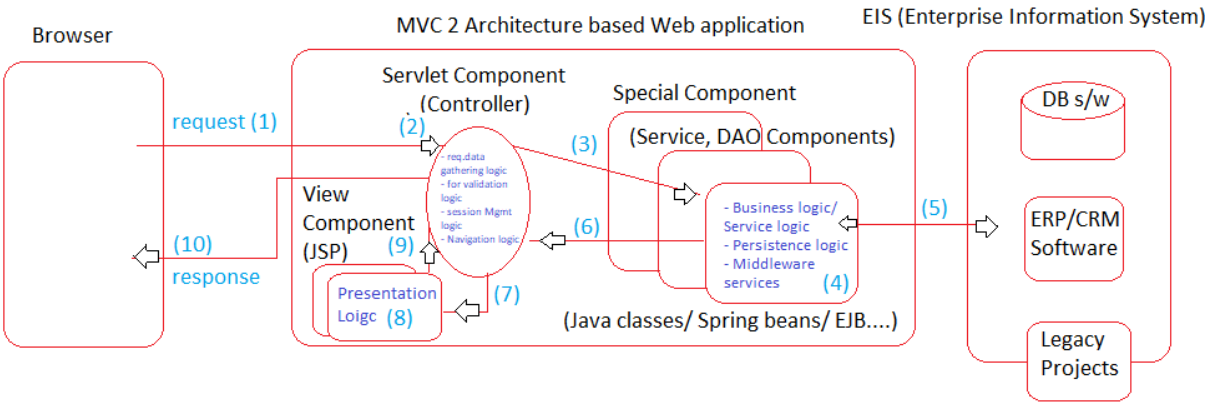
Q. What is the difference between MVC 1 and MVC 2?

Ans.

- In MVC 1, we take single component (Servlet/ JSP component) acting as both view and controller, but we take separate components for model layers.
- In MVC 2, we take separate components for view (JSP), separate component for controller (Servlet), and we also take separate component for model layer.

MVC 2 Architecture diagram

- Multiple backend software together is called EIS (Enterprise Information System).
- EIS = DB s/w + ERP s/w (Like SAP) + CRM (Siebel) + Legacy Projects
- The project that is develop using old technologies or old versions of the existing technologies is called Legacy Project.
- ERP: Enterprise Resource Plan.
- CRM: Customer Relation Management.



With respective to diagram:

1. Browser gives request.
2. The Controller Servlet Components traps and takes the request.
3. Using Navigation logic, the Controller servlet passes request to appropriate model component (Service, DAO component).
4. The Service, DAO component processes the request.
5. DAO component interacts with EIS as backend.
6. The results generated by the service component comes back to controller.
7. The controller component passes the results to view component.
8. View component formats the results using presentation logics.
9. The formatted results go to browser as response through controller components (10 point also same).

MVC 2 architecture Pros and Cons

Pros:

- Since there are multiple layers in application development, we can say that there is clean separate between logics.
- The modifications done in one-layer logics does not affect other layer logics.
- Maintenance and enhancement of the project becomes easy.
- Parallel development is possible. So, productivity is good.
- It is industry standard architecture to develop the Java based medium scale and large-scale websites.

Cons:

- Since the parallel development is possible, we need more programmers.
- Knowledge on multiple technologies is required.

Q. How parallel development is possible in MVC 2 architecture?

Ans.

- The Project leader, manager divides the team into two parts
- Part 1: Presentation tier component developers.
    - This part of the team takes the support of Servlet, JSP technologies and other UI technologies and develops view, controller component,
- Part 2: Business tier component developers.
    a. This part of the team takes the support of Spring beans or Java classes or EJB components and other model layer technologies to develop business logics and persistence logic (Service classes and DAO classes).

Note: Since these two parties can work parallelly either from same location or from two different locations, so we can say parallel development is possible.

MVC 2 architecture Rules or Principles

- MVC 2 architecture is not just working with multiple technologies in multiple layers. There are set of rules that we need to follow.
    a. Every layer is given to place bunch of logics. Just place only those logics and do not add any additional logics.
    b. There can be multiple View components, multiple model components, but it is recommended to take single controller component.
    c. All the operations must take place under the controller or

monitoring of controller servlet.

d. View components must not interact with model components directly and vice-versa. They must interact through controller servlet component.

Q. When MVC 2 architecture is there to develop web applications as the layered applications. what is need of Web application frameworks in Java?

Ans.

- If we develop MVC 2 architecture manually by using Servlet, JSP technologies then,
  a. Programmers should develop all the logics are all layers manually.
  b. Programmers should take care of navigation management (should decide which request should go which model component).
  c. Programmers should take care of view management (should decide which model component results should go which view component).
  d. Programmers should take care of data management (should worry about how to pass data among the layer and scope of data).
  e. Should remember and implement MVC 2 rules.
  f. Chance of having boilerplate code.
  and etc.

Note: To overcome from the above problems and to get abstraction on Servlet technologies in web application development we can take the support of MVC2 frameworks or Java web application frameworks they are

- Struts from Apache
- JSF from sun MS (Oracle corporation) (2)
- Webwork from Open symphony
- ADF from Oracle corporation
- Spring MVC/ Spring Boot MVC from Interface21/ pivotal team (1)
  and etc.

Advantages of developing MVC 2 architecture-based web application using Web application framework:

a. Gives readymade controller servlet i.e., we need not write controller logics manually.
b. Automatically implements maximum MVC 2 rules.
c. The readymade controller servlet can trap all or multiple requests to apply common system services like logging, auditing, security and etc.
d. FrontController Servlet itself takes care of navigation management.

e. FrontController Servlet itself takes care of view management.
f. FrontController Servlet itself takes care of Data management.
g. Lots of boilerplate code will be avoided.
and etc.

Note:

- ✓ The readymade controller servlet will be given based on "FrontController" Design Pattern. So, it is also called as FrontController Servlet.
- ✓ In Struts framework the readymade front controller Servlet is "ActionServlet".
- ✓ In JSF framework the readymade front controller Servlet is "FacesServlet"
- ✓ In Spring MVC/ Spring Boot MVC framework the Readymade front controller Servlet is "DispatcherServlet".

Q. Is MVC 1, MVC 2 and Model1 are the Design Patterns or Architectures?
Ans. These are architectures to develop the Java based web applications.

Q. What is the different between Architecture and Design Pattern?
Ans.

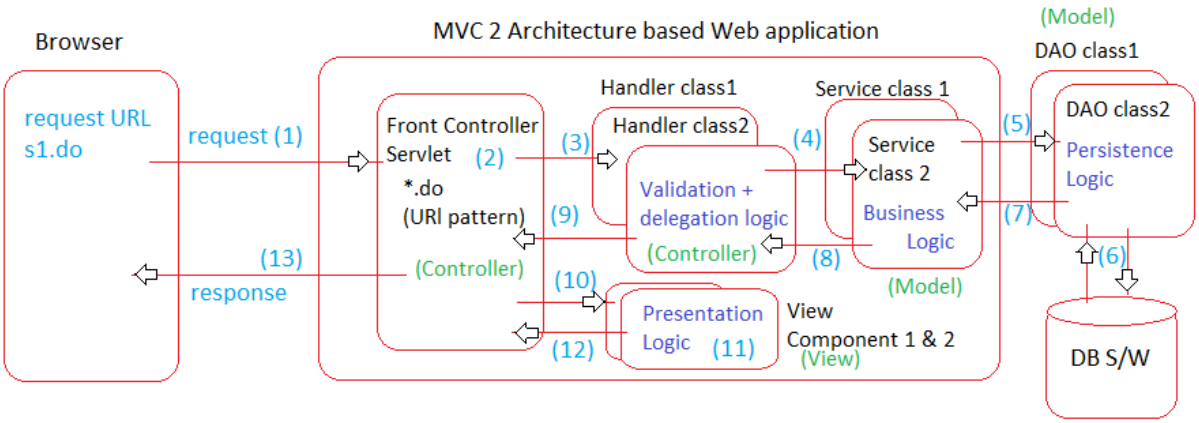
- Architecture speaks about involving multiple components and their flow of execution in the Application development.
- Design Pattern speaks about the best solution for reoccurring problem in application development.
- In the implementation of architecture, multiple design patterns will be involved.
  E.g.,
  MVC 2 architecture says how to involve in multiple components in layered web application development. In each layer multiple Design patterns will be used to solve the commonly reoccurring problems,

  In Controller Layer:
  FrontController, ApplicationController, Intercepting Filter and etc. design patterns will be applied.
  In View Layer:
  View Helper, Composite View and etc. patterns will be involved.
  In Model Layer:
  Service Delegate/ Business delegate, DAO and etc. patterns will be involved.

FrontController Servlet

- The Special web component/ servlet component of MVC 2 architecture of based Java web application who traps either all requests or multiple requests to apply common system services like security, logging, auditing and etc. and also takes cares of navigation management, view management, model/ data management is called FrontController Servlet.
- Navigation Management: Decides the flow among the components.
- View Management: Decides the which model component/ service class or handler class result should go to which view component (JSP or another component).
- Data/ Model Management: Talks about how to store input values (form data) and how to pass the generated results to various components by keeping them in scope.
- Once the FrontController Servlet is involved in MVC 2 architecture web application there will be only one servlet in the entire web application i.e., Front Controller Servlet.
- We keep Java classes as handler/ controller/ action classes in web application either to process the request directly or to delegate the requests to service classes by taking from FrontController Servlet.

MVC 2 Architecture + FrontController Design Pattern



- Only FrontController is Servlet component.
- Handler classes, Service classes, DAO classes are plain java classes.
- View comps are generally JSP/ HTML and etc. pages.
- To make FrontController servlet trapping multiple requests take the support of extension match or directory match URL pattern.
  E.g.,
  - *.do, *.nit, *.htm, *.com (extension match URL pattern)
  - /x/y/*, /abc/mno/*, /nit/* (directory match URL pattern)

- To make FrontController servlet trapping all requests take the support "/" URL pattern.

With respect to the diagram:

1. Browser gives request.
2. FrontController Servlet traps the request and applies the common system services.
3. FrontController Servlet delegates the request appropriate handler/ controller/ action class.
4. Handler class uses Service to process the request by using business logic and persistence logics.
5. For the process Service classes use DAO classes.
6. And DAO classes talk with DB s/w if needed.
7. DAO class return the result to service class.
8. Handler class gets results of busine logic execution from Service, DAO classes.
9. Handler class keeps the results in request scope and passes the control to FrontController Servlet.
10. FrontController Servlet delegates to the appropriate view component.
11. View component reads the results from request scope and formats the results.
12. Formatted results go to the FrontController servlet.
13. And from FrontController servlet sends the result as response to the browser.

Note: FrontController servlet makes you to handle multiple URLs based multiple requests using multiple ordinary Java classes called handler classes with the support single FrontController Servlet

Q. What is FrontController?

Ans.

- The special web component of web application that acts entry and exit point either for all requests or for multiple requests is called FrontController.
- Generally, we take Servlet component as FrontController, so we call it as FrontController Servlet.
- FrontController traps the requests, applies the system services, delegates the requests to handler classes for request processing, takes results from handler classes, passes results to view components for formatting the results and sends the results to browser as response.

- To make FrontController trapping and taking either all requests or multiple requests we need to configure or map them with "/" URL pattern or extension match or directory match URL pattern.
    - "/" URL pattern (To trap all the requests).
    - Extension or directory match URL pattern (To trap multiple requests).

Q. What is difference between FrontController and Controller/ Handler/ Action class in MVC architecture and FrontController DP based web application?

Ans.

FrontController	Controller/ Handler/ Action class
a. It is web component (generally Servlet or Servlet filter component).	a. It is java class (may be Spring bean in Spring MVC applications).
b. Traps either all requests or multiple requests and applies common System services.	b. Either directly contains request processing logics or contains logics to delegate requests to service classes.
c. Generally managed by Servlet container of the web application.	c. Managed by JVM of web Server (In Normal web applications), managed by IoC container of Spring (In Spring MVC web applications).
d. Gets request, response objects that created and given by Servlet Container.	d. Gets request, response objects that are passed by FrontController.
e. The main methods containing logics are servlet life cycle methods or convenience methods like doXxx(-,-) methods.	e. The methods that contains the business logic or delegation logics are called Handler methods.
f. Generally it is 1 per entire MVC web application.	f. Generally it is one per entire module.

Note:

- ✓ Spring MVC/ Spring boot MVC Applications the FrontController is "DispatcherServlet" which is readymade servlet component.
- ✓ Every Servlet component (either pre-defined/ readymade or user-defined) must be configured with Servlet container (i.e., class details must be given to Servlet container) and also must be mapped/ linked with URL pattern (either with "/" or with directory or extension match).

3 ways of Servlet Configuration with Servlet Container

a. Declarative Approach (using web.xml file):

- Useful if the servlet component is ready made servlet component and XML configuration are allowed in the applications.
- E.g., DispatcherServlet configuration in XML driven Spring MVC app and in XML + Annotations driven applications.

b. Annotation driven Approach (using @WebServlet):
   - Useful, if the servlet component is user-defined servlet component.
   - E.g., while developing Servlet, JSP based Model 1, MVC 1, MVC 2 architecture web applications or while developing user-defined FrontController Servlet.

c. Programmatic approach (using sc.addServlet(-,-) method):
   - Useful to configure pre-defined/ readymade servlet component in the web applications where XML configurations are not allowed.
   - E.g., In 100% code driven Spring MVC, in Spring Boot MVC applications XML configurations are not allowed, so to configure the readymade DispatcherServlet component this programmatic approach will be used either directly (manually done by programmer) or indirectly (done internally).

4 different approaches of developing Spring MVC applications

a. Declarative configuration approach (XML driven configurations)
b. Annotation configuration approach (XML + annotations driven configurations)
c. 100% code driven configuration approach
d. Spring Boot MVC

Note:

- ✓ In Declarative and Annotation configuration approach, the DispatcherServlet will be configure using web.xml.
- ✓ In 100% code and Spring Boot MVC, the DispatcherServlet will be configure using Programmatic approach/ Dynamic Servlet registration approach (directly or indirectly).
- ✓ The source of Readymade DispatcherServlet component does not contain @WebServlet annotation.
- ✓ All Spring MVC or Spring Boot MVC applications that developed in different approaches are based on MVC 2 architecture and FrontController (Dispatch Servlet) design pattern.

In Spring MVC or Spring Boot MVC applications we need to use following two annotations while developing handler classes/ controller classes

- @Controller: To mark Java class as Spring bean cum controller/ handler class
- @RequestMapping: To mark Java method of @Controller class as handler method having request path and request mode/ method (GET/ POST).

Note: As of now, the browser can give two modes of requests (GET (default), POST).

LoginOperationsController.java

```
@Controller
public class LoginOperationsController {

        @RequestMapping (value="/login", method=RequestMethod.GET)
        public String login (parameters ....) {
                ……..
                …….    //direct business logic or logic to delegate the request
                ……..                to service class
        }

}
```

Note: Handler methods in Controller classes can have flexible signatures i.e., we have multiple options for Return types and multiple options for parameters types and also method names programmer choice.

The possible Parameter types for Handler methods

Controller method argument	Description
WebRequest, NativeWebRequest	Generic access to request parameters and request and session attributes, without direct use of the Servlet API.
javax.servlet.ServletRequest, javax.servlet.ServletResponse	Choose any specific request or response type.

javax.servlet.http.HttpSession	Enforces the presence of a session. As a consequence, such an argument is never null.
javax.servlet.http.PushBuilder	Servlet 4.0 push builder API for programmatic HTTP/2 resource pushes.
java.security.Principal	Currently authenticated user possibly a specific Principal implementation class if known.
HttpMethod	The HTTP method of the request.
java.util.Locale	The current request locale, determined by the most specific LocaleResolver available.
java.util.TimeZone + java.time.ZoneId	The time zone associated with the current request, as determined by a LocaleContextResolver.
java.io.InputStream, java.io.Reader	For access to the raw request body as exposed by the Servlet API.
java.io.OutputStream, java.io.Writer	For access to the raw response body as exposed by the Servlet API.
@PathVariable	For access to URI template variables.
@MatrixVariable	For access to name-value pairs in URI path segments.
@RequestParam	For access to the Servlet request parameters, including multipart files. Parameter values are converted to the declared method argument type.
@RequestHeader	For access to request headers. Header values are converted to the declared method argument type.

@CookieValue	For access to cookies. Cookies values are converted to the declared method argument type.
@RequestBody	For access to the HTTP request body. Body content is converted to the declared method argument type by using HttpMessageConverter implement ations.
HttpEntity<B>	For access to request headers and body. The body is converted with an HttpMessageConverter.
@RequestPart	For access to a part in a multipart/form-data request, converting the part's body with an HttpMessageConverter.
java.util.Map, org.springframework.ui.Model, org.springframework.ui.Model Map	For access to the model that is used in HTML controllers and exposed to templates as part of view rendering.
RedirectAttributes	Specify attributes to use in case of a redirect (that is, to be appended to the query string) and flash attributes to be stored temporarily until the request after redirect.
@ModelAttribute	For access to an existing attribute in the model (instantiated if not present) with data binding and validation applied.
Errors, BindingResult	For access to errors from validation and data binding for a command object (that is, a @ModelAttribute argument) or errors from the validation of a @RequestBody or @RequestPart argu ments. You must declare an Errors,

	or BindingResult argument immediately after the validated method argument.
SessionStatus + class - level @SessionAttributes	For marking form processing complete, which triggers clean-up of session attributes declared through a class-level @SessionAttributes annotation.
UriComponentsBuilder	For preparing a URL relative to the current request's host, port, scheme, context path, and the literal part of the servlet mapping.
@SessionAttribute	For access to any session attribute, in contrast to model attributes stored in the session as a result of a class-level @SessionAttributes declaration.
@RequestAttribute	For access to request attributes.

Note:

- ✓ Green color type parameters will be used in Spring MVC applications, and other parameters are useful in Spring Rest application
- ✓ Spring Rest is extension of Spring MVC
- ✓ Spring MVC/ Spring Boot MVC for web application development.
- ✓ Spring Rest is for developing Restful web Services (Distributed app).

The possible Return types of Handler methods

Controller method return value	**Description**
@ResponseBody	The return value is converted through HttpMessageConverter implementations and written to the response.
HttpEntity<B>, ResponseEntity<B>	The return value that specifies the full response (including HTTP headers and body) is to be converted

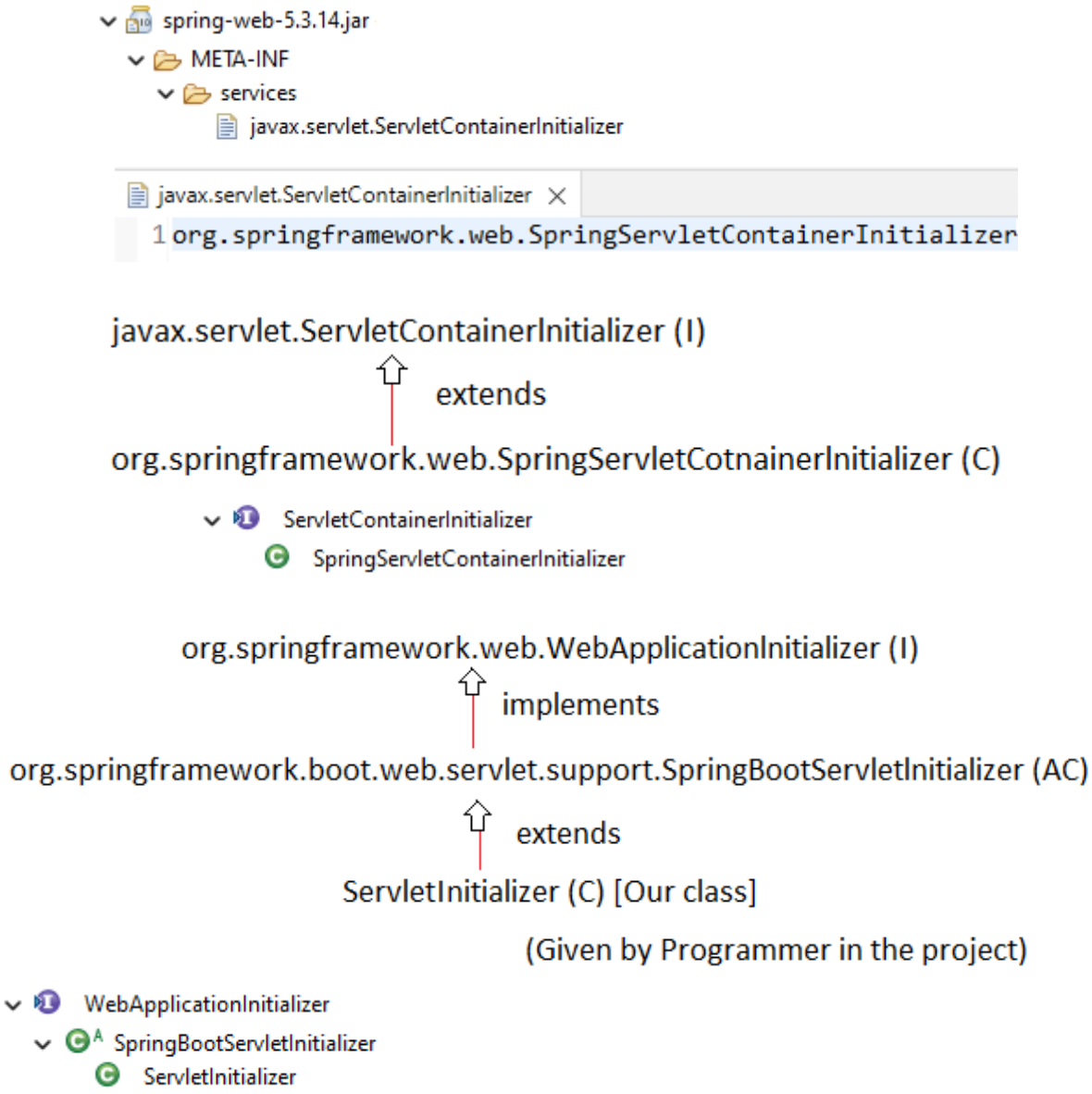
	through HttpMessageConverter implementations and written to the response.
HttpHeaders	For returning a response with headers and no body.
String	A view name to be resolved with ViewResolver implementations and used together with the implicit model - determined through command objects and @ModelAttribute methods.
View	A View instance to use for rendering together with the implicit model - determined through command objects and @ModelAttribute methods.
java.util.Map, org.springframework.ui.Model	Attributes to be added to the implicit model, with the view name implicitly determined through a RequestToViewNameTranslator.
@ModelAttribute	An attribute to be added to the model, with the view name implicitly determined through a RequestToViewNameTranslator.
ModelAndView object	The view and model attributes to use and, optionally, a response status.
void	A method with a void return type (or null return value) is considered to have fully handled the response if it also has a ServletResponse, an OutputStream argument, or an @ResponseStatus annotation.

DeferredResult<V>	Produce any of the preceding return values asynchronously from any thread - for example, as a result of some event or callback.
Callable<V>	Produce any of the above return values asynchronously in a Spring MVC-managed thread.
ListenableFuture<V>, java.util.concurrent.CompletionStage<V>, java.util.concurrent.CompletableFuture<V>	Alternative to DeferredResult, as a convenience (for example, when an underlying service returns one of those).
ResponseBodyEmitter, SseEmitter	Emit a stream of objects asynchronously to be written to the response with HttpMessageConverter implementations. Also supported as the body of a ResponseEntity.
StreamingResponseBody	Write to the response OutputStream asynchronously. Also supported as the body of a ResponseEntity.
Reactive types - Reactor, RxJava, or others through ReactiveAdapterRegistry	Alternative to DeferredResult with multi-value streams (for example, Flux, Observable) collected to a List.

Note: Green color types are used as the handler method return type in Spring MVC web application remaining all are given for Spring Rest.

Q. In Spring Boot MVC XML Configuration will not be used and the Predefined DispatcherServlet class does not contain @WebServlet annotation then How DispatcherServlet is registered/ configured with Servlet Container during the deployment of the web application?

Ans. It internally uses Programmatic approach/ Dynamic Servlet Registration approach using sc.addServlet(-,-) to register DispatcherServlet with Servlet Container having "/" as the mapping URL pattern.



Note: Generally, the ServletInitializer class creates automatically in Spring Boot starter which is taken as war type web application by adding Spring web starter.

1. Programmer deploys the Spring Boot MVC web application in Web Server.
2. The Servlet Container verifies the deployment directory structure and notices that there is no web.xml file.
3. Creates ServletContext object.
4. Servlet Container searches for "javax.servlet.ServletContainerInitializer" file in all META-INF\services folder of all jar files that are added to WEB-INF\lib folder and finds in spring-web-<ver>.jar.
5. Gets "org.springframework.web.SpringServletContainerInitializer" class name from that file.

6. Loads the class and creates the object and calls onStartup (-) having ServletContext object as the argument value.
7. This onStartup (-) method searches for class in the project implementing org.springframework.web.WebApplicationInitializer (I) and finds the programmer supplied "ServletInitializer" class.
8. Loads this "ServletInitializer" class, creates the object and calls onStartup(-) having ServletContext object as the argument value.
9. The super class (SpringBootServletInitializer) onStartup(-) method executes having logics
    a. To create WebApplicaitonContext container (IoC container) taking main class as the @Configuration class.
    b. To create DispatcherServlet class object having IoC container
    c. To register DispatcherServlet object with Servlet Container calling sc.addServlet(-) and mapping that servlet with "/" URL pattern and also enabling LoadOnStartup with value "1".

Spring Boot MVC Flow/ Spring MVC Flow

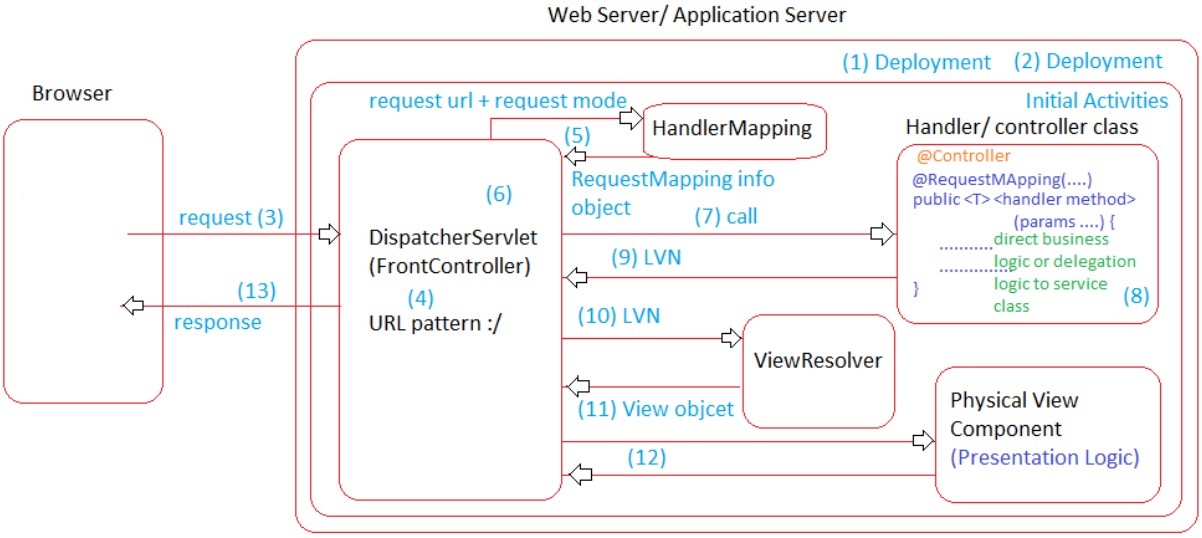
- Spring MVC/ Spring Boot MVC is designed around DispatcherServlet which is a pre-defined FrontController Servlet component.
- All the activities in Spring MVC/ Spring Boot MVC takes place under the control or monitoring of DispatcherServlet component.

Note:

- ✓ DispatcherServlet (DS) takes the support of HandlerMapping component to map/ link the given request with Handler method of Controller class. DS gives request path and request mode to HandlerMapping component and gets RequestMapping info object having the controller class bean id and Handler method signature.
- ✓ DispatcherServlet (DS) takes the support of ViewResolver component to map given request related results to one View component and gets View object having the physical view component name and location (like JSP file name and location).
- ✓ In Spring MVC/ Spring Boot MVC View component is an abstract entity i.e., we can take anything as View component like JSP files, HTML files, Spring beans, Freemarker components, Velocity components and etc.
- ✓ DS is managed by ServletContainer whereas the remaining components like HandlerMapping, ViewResolver, Controller classes, and etc. Spring beans are managed by the DS started IOC container.
- ✓ All web applications deployed in Web server/ Application server will use same Servlet container and JSP container, whereas every deployed

Spring MVC/ Spring Boot MVC web applications gets its own DispatcherServlet created IoC container.

- ✓ 1 Servlet container, 1 JSP container per Web server/ Application server.
- ✓ 1 DispatcherServlet component per Web application.
- ✓ 1 DispatcherServlet created IoC container per Web application.



With respect to the diagram:

1. Programmer deploys Spring MVC/ Spring Boot MVC application in web server or application server.
2. Deployment activities takes place as discusses earlier like IoC container creation, DispatcherServlet registration with Servlet container, Pre-instantiation of singleton scope Spring beans like controller classes, HandlerMapping, ViewResolver, Service classes, DAO classes and etc. in this process the necessary dependency injections also takes place.
3. Browser gives request to the deployed Spring MVC/ Spring Boot MVC web application.
4. The FrontController Servlet (DispatcherServlet) traps the request and applies the common system services on the request like logging, auditing, tracking and etc.
5. DispatcherServlet handovers the request to HandlerMapping component to map given incoming request with handler method of Handler/ Controller class and gets RequestMapping object from HandlerMapping component having Controller class bean id and Handler Method signature (uses lots of Reflection API internally).
6. DS takes Controller class bean id from the received RequestMapping Info objects and gets Controller class object from DS created IoC container. DS also prepares the necessary objects based on the signature of Handler method collected from the RequestMapping Info object.

7. DS calls handler method having necessary objects as the arguments on the above received Controller class object.
8. The handler method of Controller either directly process the request or takes the support Service, DAO classes to process the request and keeps the results in scope (generally request scope).
9. The handler method of controller class returns LVN (Logical View Name) back to DS.
10. DS gives LVN to ViewResolver component.
11. ViewResolver maps/ links LVN to physical view component and returns View object having physical view name and location.
12. DS gets physical view name and location from the received View object and sends the control to physical view component using rd.forward(-,-) to format the results gathered from request scope using presentation logics, these formatted results go to back to DS.
13. DS sends formatted results back to browser as response.

HandlerMapping

- This component takes the incoming requests through DispatcherServlet and maps that request with appropriate handler method of appropriate Controller class by matching incoming request URL path with handler method request path using Reflection API and returns RequestMapping Info object back to DispatcherServlet having the mapped controller class bean id and handler method signature.
- All HandlerMapping components are the classes implementing org.springframework.web.servlet.HandlerMapping (I).
- Generally, we work with readymade HandlerMapping classes,
    a. BeanNameUrlHandlerMapping (Default in XML driven configuration)
    b. RequestMappingHandlerMapping (Default in Annotation driven configuration, 100% code driven configuration, Spring Boot MVC)
    c. ControllerClassNameHandlerMapping
    d. SimpleUrlHandlerMapping
    and etc.

ViewResolver

- This component maps the received logical view name with physical view component and returns View object having the name and location of physical view component back to DispatcherServlet. It can collect inputs from either from XML file (In XML driven configuration) or from

Properties file (In Spring Boot application).

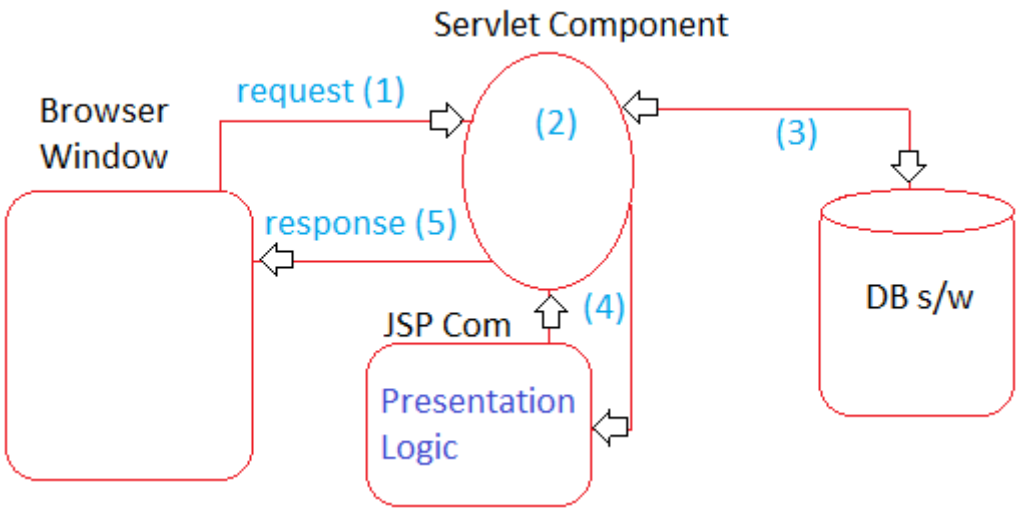
- ViewResolver components are the classes implementing org.springframework.web.servlet.ViewResolver (I).
- Generally, we work with readymade ViewResolver classes,
    a. UrlBasedViewResolver
    b. InternalResourceViewResolver (Default in Spring Boot MVC)
    c. ResourceBundleViewResolver
    d. XmlViewResolver
    e. TilesViewResolver
    f. FreeMarkerViewResolver
    g. BeanNameViewResolver
    and etc.

Note: No Default ViewResolver in XML driven configurations, Annotation driven configuration, and 100% code driven configuration.

- The "InternalResourceViewResolver" is capable of taking the JSP or HTML or Servlet component placed in private area as the View components though HTML, JSP components are not configured having URL pattern. In Spring MVC or Spring Boot MVC application we generally prefer taking "JSP" components as the view component and it even recommends to keep those JSP components in private area.
- Outside "WEB-INF" area of the web application is called "public" area of the web application. The request to public area components can be given directly without giving any mapping details to Servlet Container.
- Inside "WEB-INF" area of the web application is called "private" area of the web application. The request to private area components must be given only after giving mapping details to Servlet Container.

Advantages of placing JSP components in Private area (Inside WEB-INF folder) of the Web application:

a. Helps to hide the technology of the web application because the JSP file name does not appears in the request URLs, so same thing does not appear in browser address bar. This helps to protect the application from Hackers and Jackers.
b. Protects the source code of the JSP components from outsiders.
c. If the JSP component is developed to display request scope data given by Servlet component, then direct request to JSP component by keeping JSP in public area may give null values or ugly values. So, to stop that place JSP components in private area.



- If our Spring MVC/ Spring Boot MVC apps keeps JSP components in private area like (WEB-INF\pages folder or WEB-INF\jsp folder) then the InternalResourceViewResolver needs 3 details to locate those JSP components
    a. Location of JSP components as the prefix info (/WEB-INF/pages/ or /WEB-INF/jsp)
    b. Suffix of JSP components (the .jsp extension)
    c. File name as the LVN

```
WEB-INF
    I--> pages
        I--> abc.jsp
```

InternalResourceViewResolver needs
- /WEB-INF/pages/ as the prefix (Location)
- .jsp as the suffix (View technology)
- abc as the LVN (Logical name/ file name)
  <prefix> + <LVN> + <suffix>
  /WEB-INF/pages/abc.jsp

Note:

✓ While working with InternalResourceViewResolver we need to take the "JSP" file name as the LVN and we can supply prefix and suffix details from application.properties file of Spring Boot MVC application.

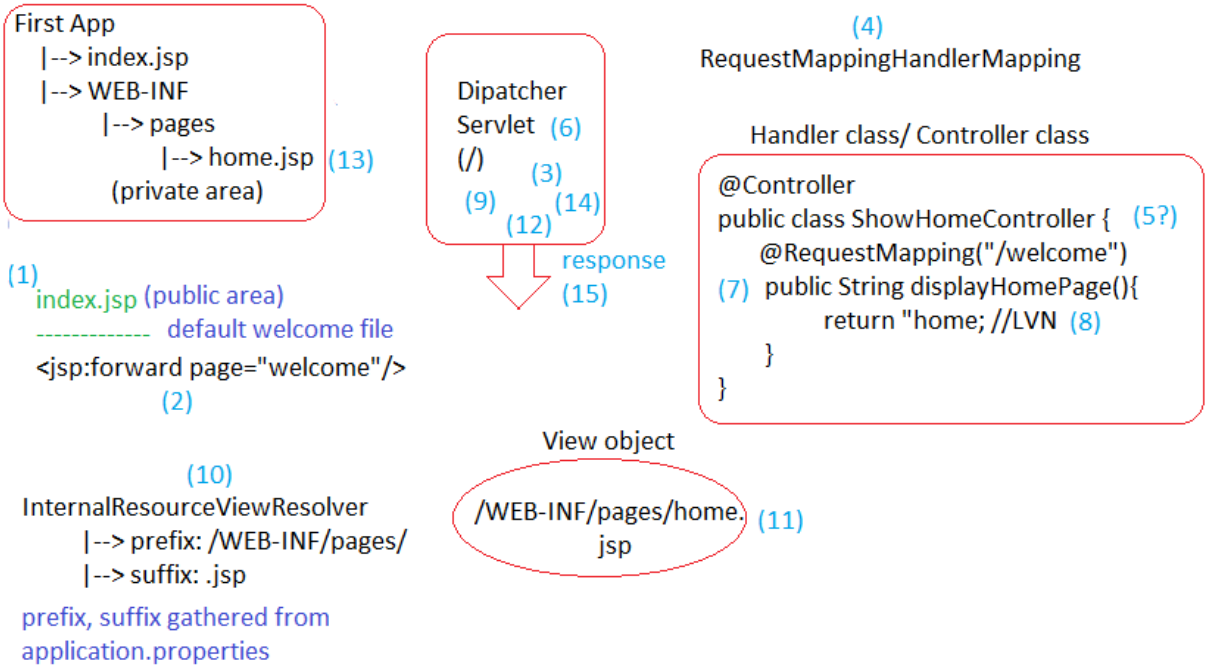
application.properties
spring.mvc.view.prefix=/WEB-INF/pages/ (Location)
spring.mvc.view.suffix=.jsp (Technology)

✓ In Spring Boot MVC applications the following components comes automatically so we need not to develop them,
    a. DispatcherServlet

b. IoC container
c. HandlerMapping
d. ViewResolver

Story Board of First Spring Boot MVC application

- To display Private area JSP component/ pages as home page.



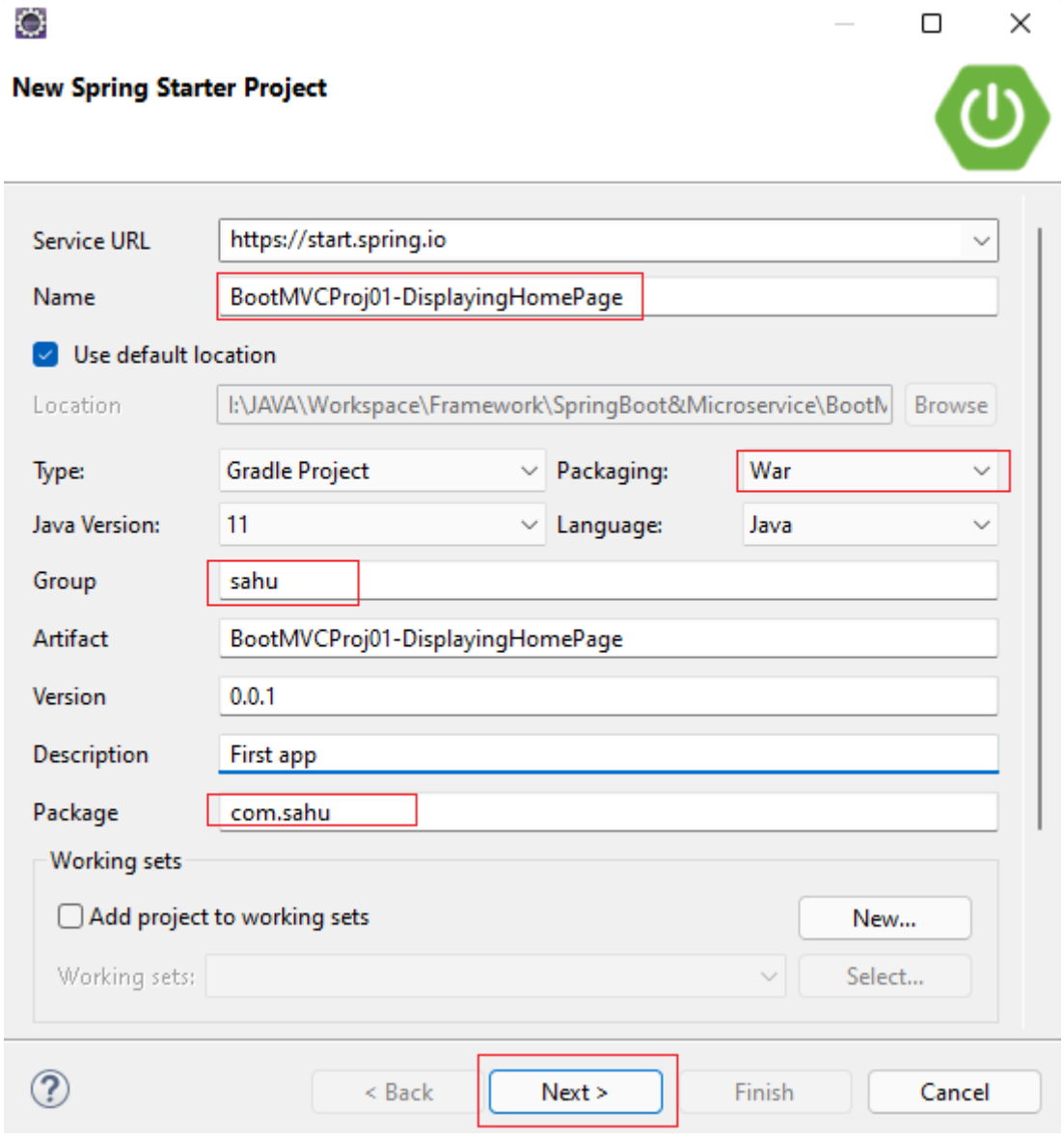
- A Spring Boot MVC application can be developed as standalone web application or deployable web application in Web Server/ Application server.
- The Spring Boot MVC standalone web application is nothing but the web application that use embedded server (default is tomcat), so no need of arranging any external server separately (Good in Development, Testing environment but not in UAT and Production environment).
- This application comes as jar file. Here Spring Application.run(-) itself creates the Embedded Tomcat server along with IoC container creation and DispatcherServlet registration. Here no need of separate ServletInitializer class.
- The Spring Boot MVC Deployable web application is nothing but the web application that is deployable in any web server as normal war file.
- Here separate ServletInitializer class is required extending from SpringBootServletInitializer class to create IoC container and to register DispatcherServlet dynamically.
- Here the created IoC container is

AnnotationConfigWebApplicationContext type IoC container.

Procedure to create First Spring Boot MVC application

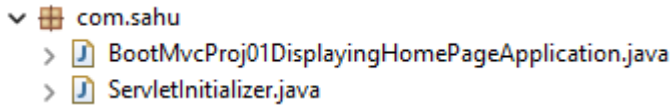
Step 1: Create Spring Boot starter project by taking the packing type as "war" and by also selecting the following starters.

- Spring Web



- The created starter project gives two ".java" files in default root package "com.sahu".
  a. Main class cum Configuration class having @SpringBootAppliocation
     (Its main (-) is not required if we running our app deployable web application using External tomcat server/ any other server)

b. ServletInitializer class extending SpringBootServletInitializer (This is not required if you are running the Spring Boot MVC app as standalone web application using embedded server).



Step 2: Add Embedded tomcat jasper jar file to the build.gradle or pom.xml file. [Collect the dependency]

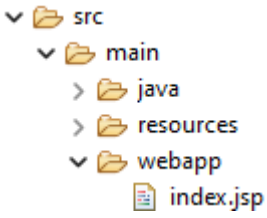
- This will be used by Embedded tomcat server to translate JSP pages into equivalent Servlet components (This is not required if you are running the application as deployable web application in external server).

Step 3: Write following entries in application.properties file to support view resolver and to decide the embedded server's port number.

application.properties

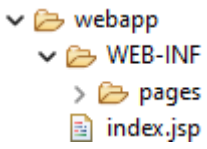
```
#ViewReoslver
spring.mvc.view.prefix=/WEB-INF/pages/
spring.mvc.view.suffix=.jsp

#Server port number
server.port=4041 (required only for standalone mode of execution)
```

Step 4: Add "index.jsp" to generate implicit request using <jsp:forward> tag.



Step 5: Develop the Controller/ Handler class.

Step 6: Develop the home.jsp file in WEB-INF/pages folder (you have to create the folders under webapp folder).



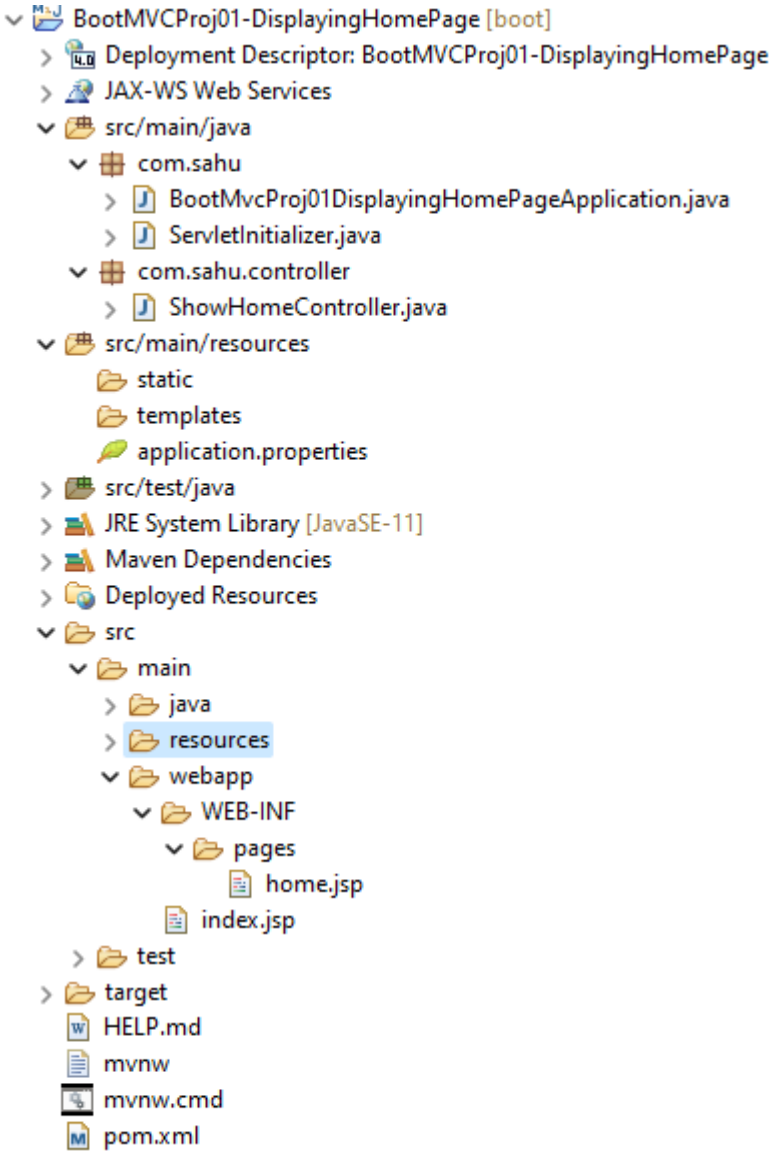
Step 7: Run the application in both modes.

a. Run As Server option (To run the application as deployable web

application in the externally configuration Tomcat server with eclipse IDE).

b. Run As Java app or Spring Boot app (To run the app as standalone web application).

Directory Structure of BootMVCProj01-DisplayingHomePage:



- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also.
- Following Staters are needed so choose them during project creation.

  X Spring Web

- And add tomcat embedded jasper jar in pom.xml if you want to Run as standalone web application.
- Then place the following code with in their respective files.

application.properties

```
#ViewResolver
spring.mvc.view.prefix=/WEB-INF/pages/
spring.mvc.view.suffix=.jsp

#Embedded Server port number
server.port=5050
```

index.jsp

```
<%@ page isELIgnored="false"%>

<jsp:forward page="welcome"/>
```

home.jsp

```
<%@ page isELIgnored="false"%>

<h1 style="color: red; text-align:center;">Welcome to Spring Boot
MVC</h1>
```

ShowHomeController.java

```
package com.sahu.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class ShowHomeController {

	@RequestMapping("/welcome")
	public String showHomePage() {
		return "home";
	}

}
```

- You can Run as Spring Boot or Java application for that we need not to do anything you can directly run but if you want to Run as Run-on Server then you have to configure external tomcat.

Note:

- ✓ Embedded tomcat server is not taking index.jsp as the default welcome file but external tomcat server takes it.
- ✓ If we run the Spring Boot MVC web application as the standalone app the SpringApplication.run(-) method takes care of multiple actives,
  a. Creating Embedded Tomcat server
  b. Creating IoC container
  c. Creating DispatcherServlet object having IoC container
  d. Creating Error Filter classes objects
  e. Registering DispatcherServlet, Error Filters dynamically with ServletContainer.
  and etc.

By Default, the standalone Spring Boot MVC apps run without context path to provide context path for those applications we need to add one special entry in application.properties.

application.properties

```
#Context path for standalone execution
server.servlet.context-path=/FirstApp
```

Old request URL: http://localhost:5050/index.jsp
New request URL: http://localhost:5050/FirstApp/index.jsp

- In order to get welcome page/ home page of the Spring Boot MVC app without typing index.jsp in the browser address bar (standalone app) or to avoid index.jsp (in both modes of Spring Boot MVC app) we need to take the handler method that gives LVN of home page having request path "/".

ShowHomeController.java

```
public class ShowHomeController {

	@RequestMapping("/")
	public String showHomePage() {
		return "home";
	}

}
```

- Request URL: http://localhost:5050/FirstApp/
  (We can remove index.jsp file from the project)

Note:

- ✓ One controller class can have any number of handler methods with unique request path + request mode.
- ✓ Only the standalone execution of Spring Boot MVC app takes the configuration context path from application.properties file. The deployable web application execution in external server takes the project name as the context path.
- ✓ If you are running Spring Boot MVC app as deployable app in external server then you can comment main (-) of @SpringBootApplication class.
- ✓ If you are running Spring Boot MVC app as standalone app using embedded server then we can remove ServletInitiazer.java file.

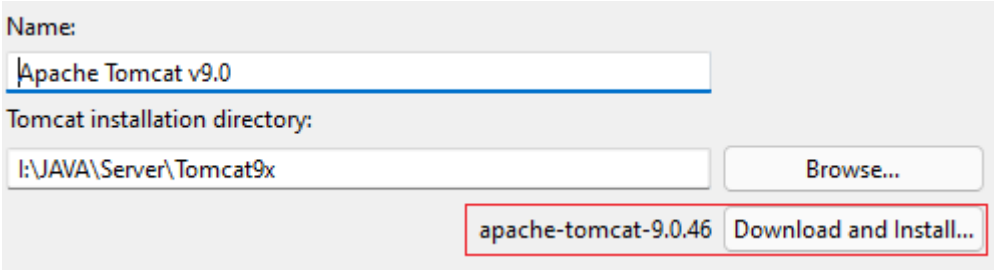
Second application Story board

- Getting Wish Message based on current hour of day.

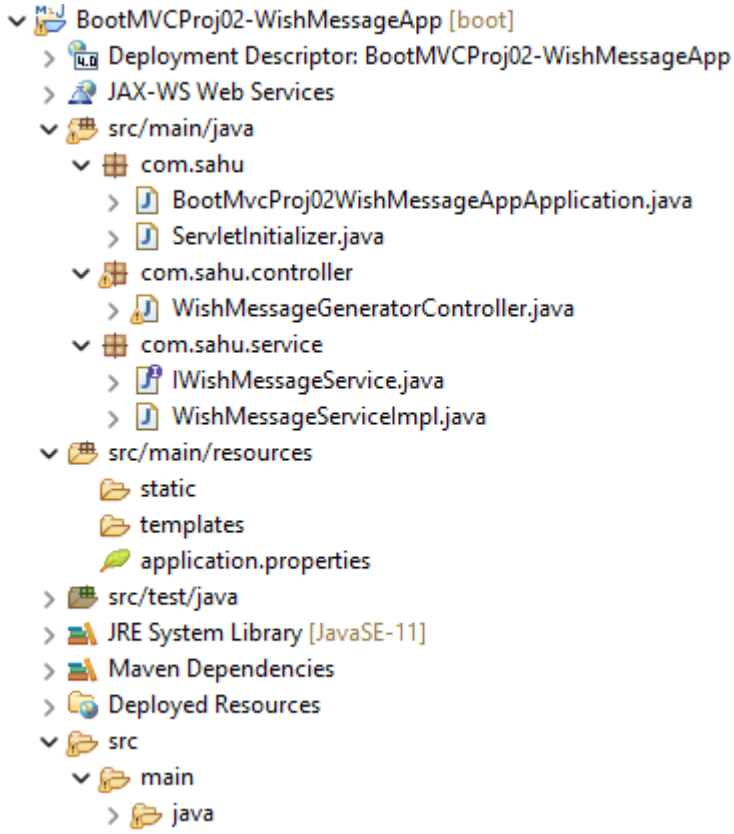
- ${wMSg}: Searches and reads the attribute value form different scopes.
- In an order page --> request -->session --> application.

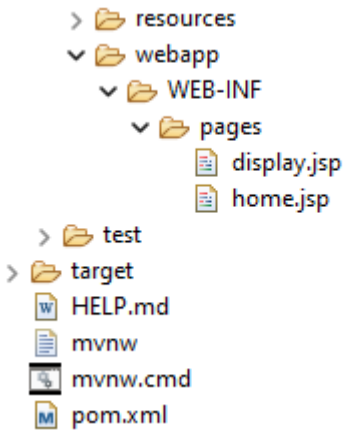
Note:

- ✓ Do not use Tomcat 9.0.21 version server or your choice version server as the external tomcat server in eclipse IDE, because for every eclipse version there is a specified tomcat server version is there you have to use that same version or its above version is good.
- ✓ For Spring Boot MVC application when you use "/" as request path it is not working when you use lower version then the required version of tomcat.
- ✓ You will get the required version of tomcat when you go to add server in eclipse. Windows - Preference - Server - Runtime Environments - Add



Directory Structure of BootMVCProj02-WishMessageApp:





- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also.
- Use same Spring Web starter during project creation.
- Copy the required Tomcat embedded jasper jar (for embedded tomcat) and application.properties from previous project.
- Then place the following code with in their respective files.

home.jsp

```
<%@ page isELIgnored="false"%>

<h1 style="color: red; text-align: center;">Home Page</h1> <br><br>
<h3 style="color: blue; text-align: center;"><a href="wish">Get Wish
message</a></h3>
```

display.jsp

```
<%@ page isELIgnored="false"%>

<h1 style="color: red; text-align: center;">Result Page</h1>
<h1 style="color: red; text-align: center;">Wish Message is  :
${wishMsg}</h1> <br><br>
<a href="./">Home</a>
```

IWishMessageService.java

```
package com.sahu.service;

public interface IWishMessageService {
        public String generateWishMessage();
}
```

WishMessageServiceImpl.java

```
package com.sahu.service;

import java.time.LocalDateTime;

import org.springframework.stereotype.Service;

@Service("wishService")
public class WishMessageServiceImpl implements IWishMessageService {

	@Override
	public String generateWishMessage() {
		LocalDateTime dateTime = LocalDateTime.now();
		//get Current hour of the day
		int hour = dateTime.getHour();
		if (hour<12)
			return "Good Morning";
		else if(hour<16)
			return "Good Afternoon";
		else if(hour<20)
			return "Good Evening";
		else
			return "Good Night";
	}

}
```

WishMessageGeneratorController.java

```
package com.sahu.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

import com.sahu.service.IWishMessageService;

@Controller
public class WishMessageGeneratorController {

	@Autowired
	private IWishMessageService messageService;
```

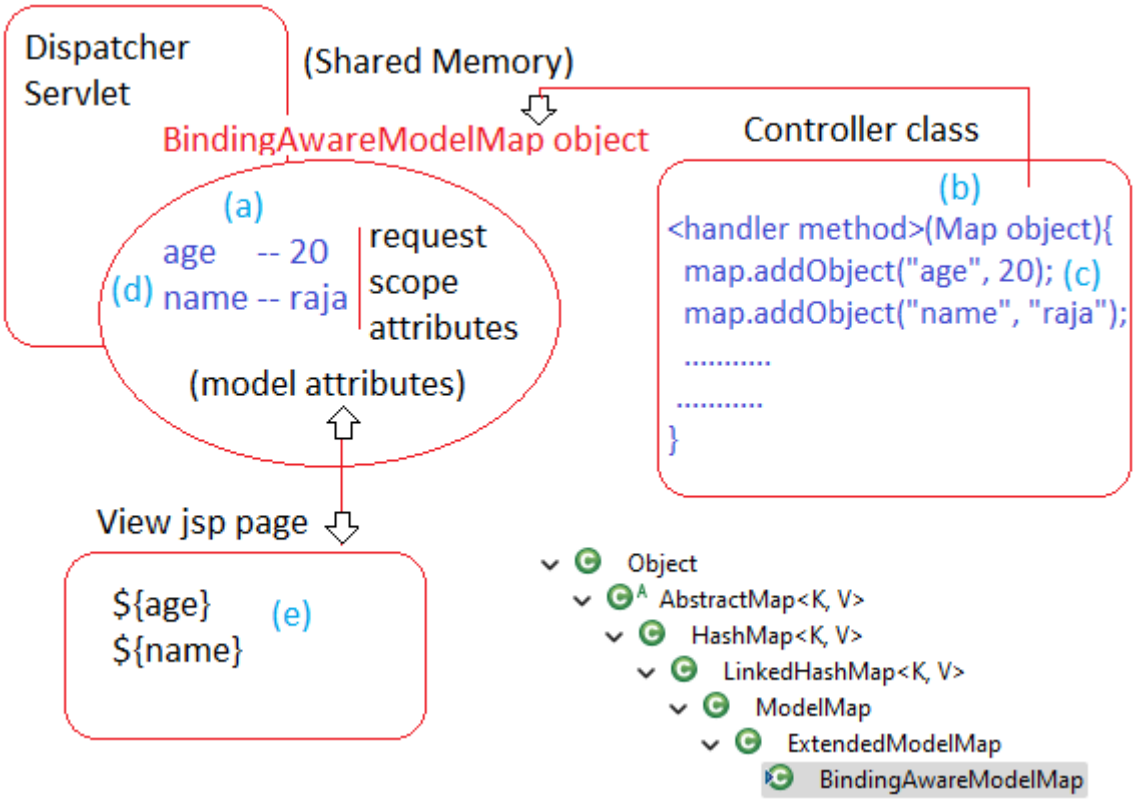
```
	@RequestMapping("/")
	public String showHome() {
		return "home";
	}

	@RequestMapping("/wish")
	public ModelAndView showWishMessage() {
		//Use service
		String wishMessage = messageService.generateWishMessage();
		//return ModelAndView object
		return new ModelAndView("display", "wishMsg",
wishMessage);
	}

}
```

Note:

- ✓ If the handler mapping method return type is String, then it returns only "LVN" back to DispatcherServlet.
- ✓ If the handler mapping method return type is ModelAndView class object then it returns "model attributes and logical view name" back to DispatcherServlet.
- ✓ Taking ModelAndView as the return type for handler method is legacy approach (Not so recommended, go for alternates).
- ✓ If DispatcherServlet receives the ModelAndView class object from the Handler method then it makes the model attributes as the request scope data (request attributes) and gives the received LVN to ViewResolver.
- ✓ ModelAndView: Model attributes + LVN
- ✓ As of now Spring Boot MVC is supporting 3 Embedded servers
  a. Tomcat (default)
  b. Jetty
  c. Under tow (earlier JBoss)

BindingAwareModelMap object (Shared Memory):

- DispatcherServlet Internally creates one shared memory called BindingAwareModelMap object to maintain model attributes having request scope.
- By exposing this BindingAwareModelMap object reference to handler methods as parameter we can add model attributes to it (default scope is request scope). So, all view components belonging to same request scop can read and use model attributes data.



WishMessageGeneratorController.java

```
@RequestMapping("/wish")
public String showWishMessage(BindingAwareModelMap map) {
        //Use service
        String wishMessage = messageService.generateWishMessage();
        map.addAttribute("wishMsg", wishMessage);
        //return LVN
        return "display";
}
```

WishMessageGeneratorController.java

```
@RequestMapping("/wish")
public String showWishMessage(ModelMap map) {
        //Use service
        String wishMessage = messageService.generateWishMessage();
        map.addAttribute("wishMsg", wishMessage);
        //return LVN
        return "display";
}
```

Note: There are so many approaches to improve the Handler method.

WishMessageGeneratorController.java

```
@RequestMapping("/wish")
public String showWishMessage(HashMap<String, Object> map) {
        //Use service
        String wishMessage = messageService.generateWishMessage();
        map.put("wishMsg", wishMessage);
        //return LVN
        return "display";
}
```

WishMessageGeneratorController.java ⭐ (recommended)

```
@RequestMapping("/wish")
public String showWishMessage(Map<String, Object> map) {
        //Use service
        String wishMessage = messageService.generateWishMessage();
        map.put("wishMsg", wishMessage);
        //return LVN
        return "display";
}
```

WishMessageGeneratorController.java

```
@RequestMapping("/wish")
public String showWishMessage(Model model) {
        //Use service
        String wishMessage = messageService.generateWishMessage();
        model. addAttribute ("wishMsg", wishMessage);
        //return LVN
        return "display";
}
```

WishMessageGeneratorController.java

```
@RequestMapping("/wish")
public void showWishMessage(Model model) {
        //Use service
        String wishMessage = messageService.generateWishMessage();
        model. addAttribute ("wishMsg", wishMessage);
}
```

Note:

- ✓ If the handler method return type is "void" it takes the incoming request path (excluding "/") as LVN by using RequestToViewNameTranslator concept internally.
  E.g., If request path of incoming request is "/wish" then it takes "wish" as LVN.
- ✓ If you are using only "InternalResourceViewResolver" in the Spring MVC application then all view components must be there in same location (prefix) and in same technology (suffix).
- ✓ To keep different view components in different locations and in different technologies take the support of multiple ViewResolver.

WishMessageGeneratorController.java

```
@RequestMapping("/wish")
public Model showWishMessage() {
        //Use service
        String wishMessage = messageService.generateWishMessage();
        //Create model attribute
        Model model = new ExtendedModelMap();
        model.addAttribute("wishMsg", wishMessage);
        return model;
}
```

- Taking Model/ ModelMap/ Map/ HashMap/ BindingAwareModelMap as the return type of the handler method is bad practice because of the following reasons,
  a. We should know the process of creating Model objects explicitly.
  b. The internally created shared memory (BindingAwareModelMap) object will be wasted.
  c. There is no control on LVN, we are forced to take incoming request URI's request path as the LVN.

- Taking Model/ ModelMap/ Map/ HashMap/ BindingAwareModelMap as the parameter type is good practice, especially taking Map<-, -> is more recommended good practice. The advantages are,
  d. Full control on LVN (we can take String return type).
  e. Creating BindingAwareModelMap object as the shared memory is taken care by DS.
  f. Creating our parameter type reference variable pointing to the shared memory is also taken care by DS.

g. The internally created shared memory will not be wasted.
h. By Taking parameter type as Map or HashMap we make handler method as non-invasive.

Note: Taking the return of Handler method as ModelAndView is legacy approach and not recommended to use because it makes code as invasive code.

WishMessageGeneratorController.java

```
@RequestMapping("/wish")
public Map<String, Object> showWishMessage() {
        //Use service
        String wishMessage = messageService.generateWishMessage();
        //Create Map object
        Map<String, Object> map = new HashMap<>();
        map.put("wishMsg", wishMessage);
        return map;
}
```

- We can make handler method sending its output to browser directly as response without involving DS, ViewResolver, View components.
- For this we need to get HttpServletResponse object as the parameter of Handler method and we should return "null" (if the return type is other than void) or return nothing (if the return type is not void).

WishMessageGeneratorController.java

```
@RequestMapping("/wish")
public void showWishMessage(HttpServletResponse response)
throws Exception {
        // Use service
        String wishMessage = messageService.generateWishMessage();
        //get PrintWriter pointing to response object
        PrintWriter pw = response.getWriter();
        response.setContentType("text/html");
        pw.println("<b>Wish message : </b>"+wishMessage);
        return;
}
```

Note: This kind of handler methods are so useful while performing file downloading activities.

WishMessageGeneratorController.java

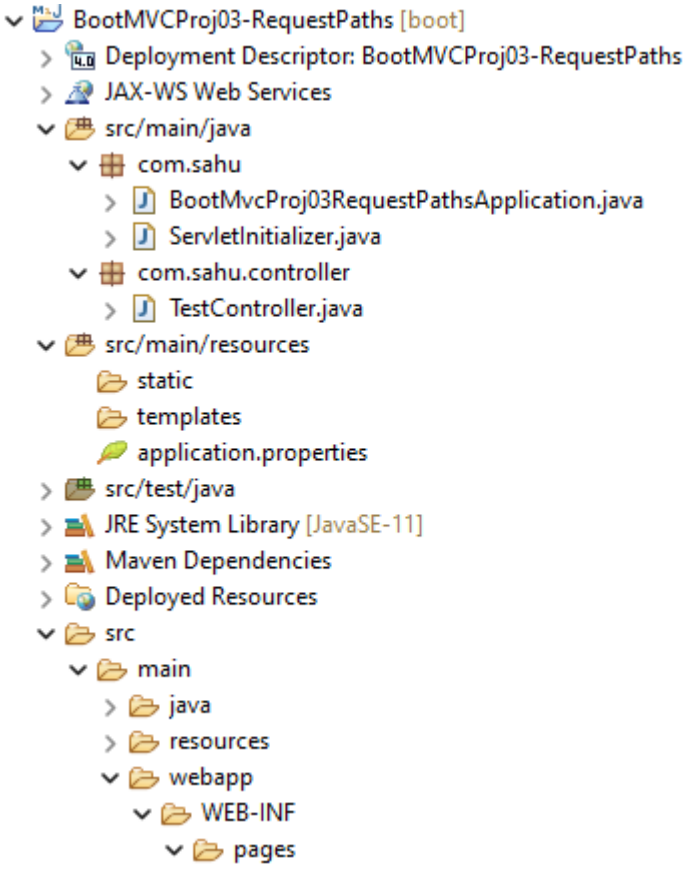
```
        @RequestMapping("/wish")
        public String showWishMessage(HttpServletResponse response)
throws Exception {
                // Use service
                String wishMessage = messageService.generateWishMessage();
                //get PrintWriter pointing to response object
                PrintWriter pw = response.getWriter();
                response.setContentType("text/html");
                pw.println("<b>Wish message : </b>"+wishMessage);
                return null;
        }
```

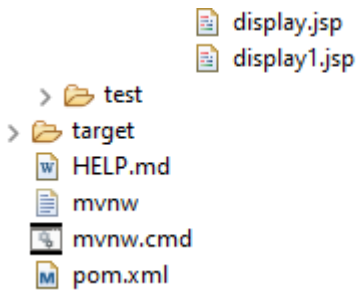
Different types of Request Path

We need to consider the following points while giving request path to handler methods of controller class

1. The request path of handler method must start with "/".
2. The request path of handler method is case sensitive.

Directory Structure of BootMVCProj03-RequestPaths:





- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also.
- Use same Spring Web starter during project creation.
- Copy the required Tomcat embedded jasper jar (for embedded tomcat) and application.properties from previous project.
- Then place the following code with in their respective files.

Display.jsp

```
<%@ page isELIgnored="false"%>

<h1 style="color: red; text-align: center;">display.jsp Page</h1>
```

display1.jsp

```
<%@ page isELIgnored="false"%>

<h1 style="color: red; text-align: center;">display1.jsp Page</h1>
```

TestController.java

```
package com.sahu.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class TestController {

	@RequestMapping("/REPORT")
	public String showReport() {
		System.out.println("TestController.showReport()");
		return "display";
	}
```

```
        @RequestMapping("/report")
        public String showReport1() {
                System.out.println("TestController.showReport1()");
                return "display1";
        }
}
```

- http://localhost:2525/BootMVCProj03-RequestPaths/REPORT (Executes showReport(-) method)
- http://localhost:2525/BootMVCProj03-RequestPaths/report (Executes showReport1(-) method)

3. Multiple methods can have same request path with different request methods/ modes like GET, POST.
   a. From browser (web client) we can give only GET or POST mode requests.
   b. URL typed in the browser address bar and hyperlink gives only "GET" mode request.
   c. Form page can generate either GET mode (default is GET) or POST mode requests.

- Create a home.jsp under WEB-INF/pages folder

home.jsp

```
<%@ page isELIgnored="false"%>

<form action="report" method="post">
        <input type="submit" value="send"/>
</form>
<br>
<a href="report">Link 1</a>
```

TestController.java

```
@Controller
public class TestController {

        @RequestMapping("/")
        public String showHome() {
                return "home";
```

```
	}

	@RequestMapping(value = "/report", method = RequestMethod.GET)
	public String showReport() {
		System.out.println("TestController.showReport()");
		return "display";
	}

	@RequestMapping(value= "/report", method =
RequestMethod.POST)
	public String showReport1() {
		System.out.println("TestController.showReport1()");
		return "display1";
	}

}
```

- Form is sends requests to showReport1(-) method whose request path is “/report” and mode is POST.
- Hyperlink <a> tag sends requests to showReport(-) method whose request path is “/report: and mode is GET.

Note: Instead of use directly use using @RequestMapping by specifying request mode GET or POST we can @GetMapping, @PostMapping annotations (given from Spring 4.x).

TestController.java

```
	@GetMapping("/report")
	public String showReport() {
		System.out.println("TestController.showReport()");
		return "display";
	}

	@PostMapping("/report")
	public String showReport1() {
		System.out.println("TestController.showReport1()");
		return "display1";
	}
```

4. One handler method can have multiple request paths.

TestController.java

```
@Controller
public class TestController {

	@GetMapping(value={"/report", "/report1", "/report2"})
	public String showReport() {
		System.out.println("TestController.showReport()");
		return "display";
	}

}
```

- All the 3 request URLs send request to same handler method that is showReport(-) method.
  http://localhost:2525/BootMVCProj03-RequestPaths/report
  http://localhost:2525/BootMVCProj03-RequestPaths/report1
  http://localhost:2525/BootMVCProj03-RequestPaths/report2

5. Taking @RequestMapping without specifying request path, it takes "/" as the default request path.

TestController.java

```
@Controller
public class TestController {

	@RequestMapping
	public String showHome() {
		return "home";
	}

}
```

6. Request path + Mode together must be unique with in a controller class otherwise we will get the below exception.
   Caused by: java.lang.IllegalStateException: Ambiguous mapping. Cannot map 'testController' method com.sahu.controller.TestController#showReport1()to {GET [/report]}:

There is already 'testController' bean method com.sahu.controller.TestController#showReport() mapped.

TestController.java

```
@Controller
public class TestController {

	@GetMapping("/report")
	public String showReport() {
		System.out.println("TestController.showReport()");
		return "display";
	}

	@GetMapping("/report")
	public String showReport1() {
		System.out.println("TestController.showReport1()");
		return "display1";
	}

}
```

7. Spring MVC/ Spring Boot MVC app at Max two methods can be taken without request path (one with GET mode or another with POST mode).

TestController.java

```
@Controller
public class TestController {

	@GetMapping
	public String showHome() {
		return "home";
	}

	@PostMapping
	public String showHome1() {
		return "home";
	}

}
```

Note:

- ✓ Like @GetMapping, @PostMapping we also got @PutMapping, @DeleteMapping, @PatchMapping and etc. annotations but these given to use in Spring Rest application i.e., can't be used in Spring MVC/ Spring Boot MVC applications because browser can give only GET, POST mode requests.
- ✓ Spring MVC/ Spring Boot MVC is given to develop web applications which allows only browser as the client. So, it can handle only GET, POST mode requests.
- ✓ Spring REST is enhancement of Spring MVC and given to develop Distributed applications (Restful web services). It allows different types of clients sending different modes of requests like GET, POST, PUT, DELETE and etc.

8. If two controller classes having two handler methods with same request path and request mode also then they can be differentiated using the controller classes level global request path.

- Create a DemoController.java under com.sahu.controller package.
- And place the following code with their respective file.

Problem:

TestController.java

```
@Controller
public class TestController {

	@RequestMapping("/")
	public String showHome() {
		return "home";
	}

	@GetMapping(value={"/report"})
	public String showReport() {
		System.out.println("TestController.showReport()");
		return "display";
	}

}
```

DemoController.java

```
package com.sahu.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class DemoController {

	@GetMapping(value={"/report"})
	public String generateReport() {
		System.out.println("DemoController.generateReport()");
		return "display";
	}

}
```

We will get the following exception,
Caused by: java.lang.IllegalStateException: Ambiguous mapping. Cannot map 'testController' method
com.sahu.controller.TestController#showReport()
to {GET [/report]}: There is already 'demoController' bean method com.sahu.controller.DemoController#generateReport() mapped.

Solution

- Take Global path or class level path like below.

DemoController.java

```
@Controller
@RequestMapping("/demo")
public class DemoController {

	@GetMapping(value={"/report"})
	public String generateReport() {
		System.out.println("DemoController.generateReport()");
		return "display";
	}

}
```

TestController.java

```
@Controller
@RequestMapping("/test")
public class TestController {

	@RequestMapping("/")
	public String showHome() {
		return "home";
	}

	@GetMapping(value={"/report"})
	public String showReport() {
		System.out.println("TestController.showReport()");
		return "display";
	}

}
```

- http://localhost:2525/BootMVCProj03-RequestPaths/demo/report
  Gives the request to DemoController.generateReport(-) method.
- http://localhost:2525/BootMVCProj03-RequestPaths/test/report
  Gives the request to TestController.showReport(-) method.

Note: In class level path or global path, you can't take @GetMapping and @PostMapping annotations.

9. We can inject ServletContext object, ServletConfig object, HttpSession object to Controller class through @Autowired Annotations.

DemoController.java

```
@Controller
@RequestMapping("/demo")
public class DemoController {

	@Autowired
	private ServletContext servletContext;

	@Autowired
```

```
        private ServletConfig servletConfig;

        @Autowired
        private HttpSession session;

        @GetMapping(value={"/report"})
        public String generateReport() {
                System.out.println("DemoController.generateReport()");
                System.out.println("Web application name :
"+servletContext.getContextPath());
                System.out.println("DS Logical name :
"+servletConfig.getServletName());
                System.out.println("Session id : "+session.getId());
                return "display";
        }

}
```

10. We cannot take ServletContext object, ServletConfig object as the parameters of handler method because they are global objects so go for @Autowired based injection as shown above more over they are not valid parameter types for handler methods, but we can take HttpSession type parameter in handler method.

DemoController.java

```
@Controller
@RequestMapping("/demo")
public class DemoController {

        @GetMapping(value={"/report"})
        public String generateReport(HttpSession session) {
                System.out.println("DemoController.generateReport()");
                System.out.println("Session id : "+session.getId());
                return "display";
        }

}
```

Data Binding & Data Rendering

- The process of writing input values (form data/ request parameters values) to Java class object (command class/ model class object) is called Data Binding.
- The process of giving controller generated/ gathered results/ outputs after executing business logic to view components through shared memory (BindingAwareModelMap object) is called Data Rendering.

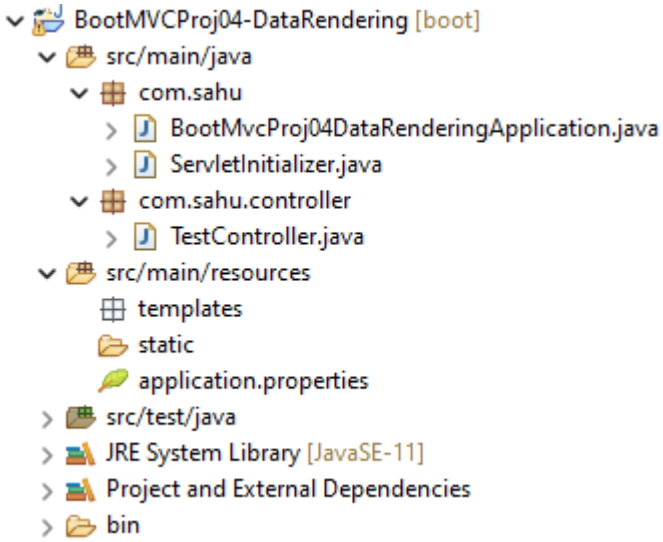
Data Rendering

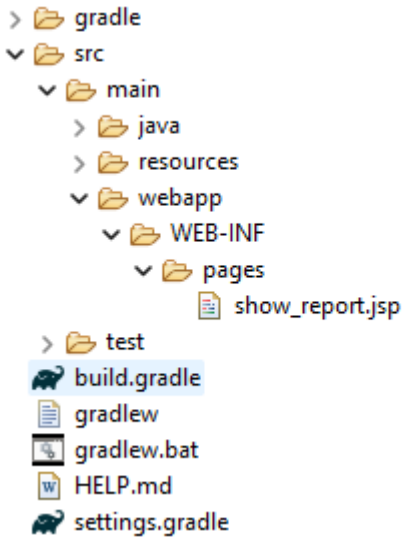
- Passing data/ results/ outputs from Controller class to view components as model attributes through Shared Memory called BindingAwareModelMap object.
- We can pass the data in following ways,
    a. Passing simple values
    b. Passing arrays and collections
    c. Passing collection of model/ Entity/ BO class objects
    d. Passing single object of model/ BO/ Entity class

Note:

- ✓ The model class is java/ java bean class which can hold data in data binding or in data rendering and it is no way related to org.springframework.ui.Model (I) implemented by BindingAwareModelMap class.
- ✓ In JSP page we can JSTL tags + EL to read model attribute values from different scope by avoiding Java code in JSP pages.

Directory Structure of BootMVCProj04-DataRendering:





- This time we are using gradle so choose project type is gradle.
- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also
- Use same Spring Web and Lombok starter during project creation.
- Copy the Tomcat embedded jasper jar (for embedded tomcat) and application.properties from previous project.
- Add JSTL jar dependency in build.gradle. [Click here]
- Then place the following code with in their respective files.

TestController.java

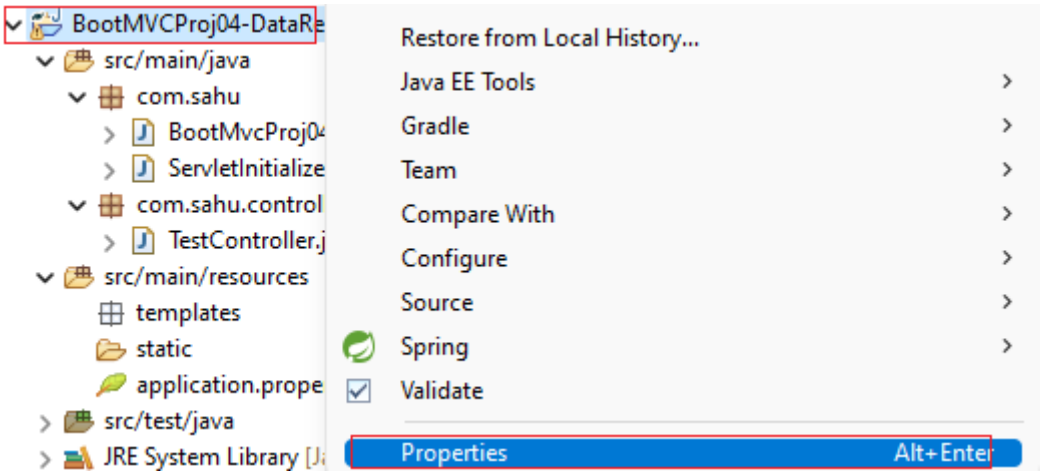
```
package com.sahu.controller;

import java.util.Map;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;

@Controller
public class TestController {
	@GetMapping("/report")
	public String  showReport(Map<String, Object> map) {
		//Add model attributes (Simple values)
		map.put("name", "Rajkumar");
		map.put("age", 40);
		map.put("address", "Hyd");
		//retrun LVN
		return "show_report";
	}
}
```

show_report.jsp

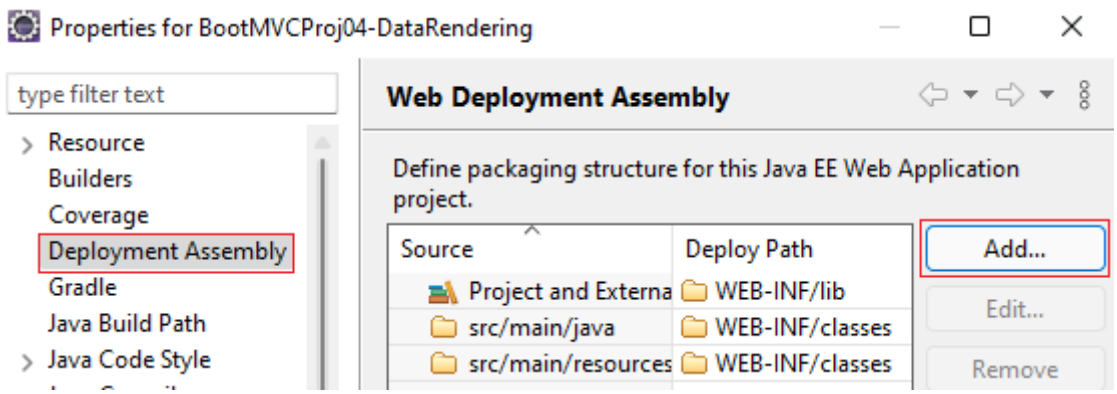
```
<%@ page isELIgnored="false" %>

<h1 style="color: red; text-align: center;">Reading Simple Values</h1>
<b>Name : ${name}</b><br>
<b>Age : ${age}</b><br>
<b>Address : ${address}</b>
```

Note: While working with gradle web application we need add webapp/WEB-INF folder folders in src/main folder manually and we need to configure src/main/webapp folder participating in deployment using "Deployment Assembly Wizard" if it is not there. For that follow the below steps,

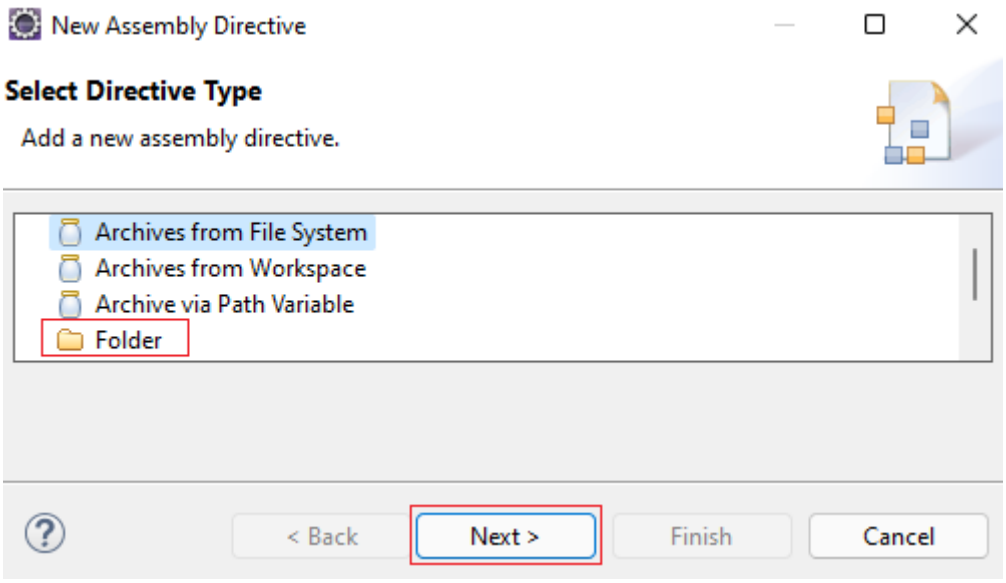
Step 1: Right click on project, then go to Properties.



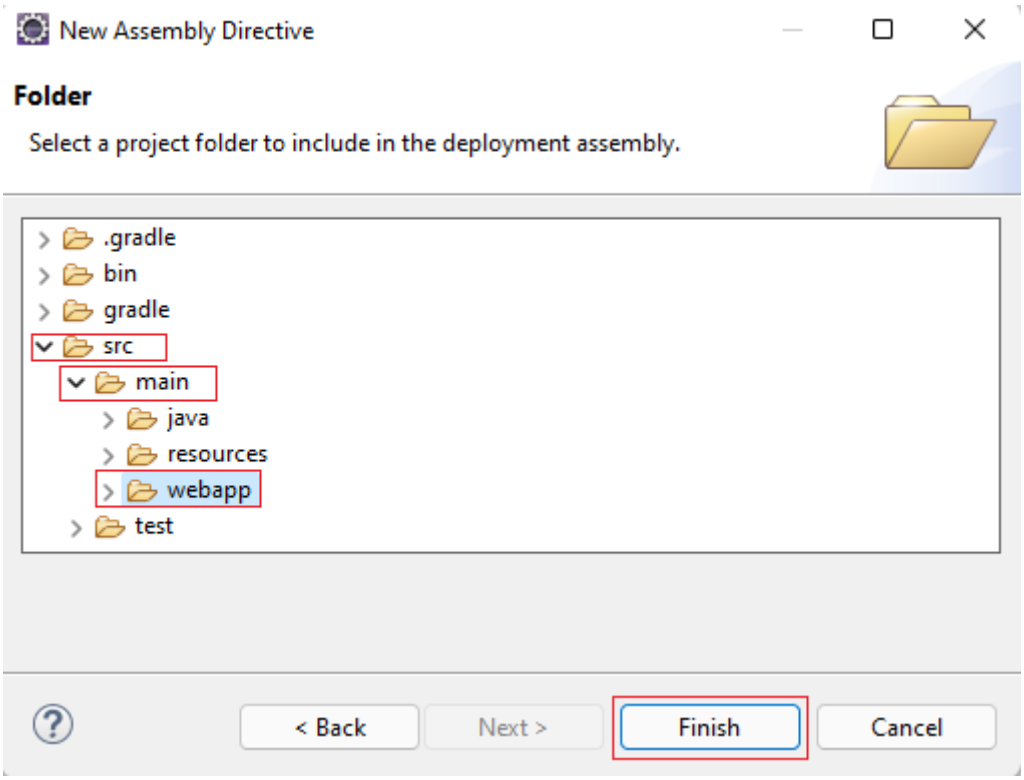
Step 2: Click on Deployment Assembly then click on Add... button.



Step 3: Choose folder option and click on Next>.



Step 4: Choose webapp folder under src/main then click on Finish after that Apply, Apply and Close.



Note: In Spring Boot project using gradle, if you want to add some external jar from MVN repository then collect the dependency from Gradle (Short) section.

Passing Arrays and Collection values from Controller class to View component in Data Rendering process

TestController.java

```
@Controller
public class TestController {

	@GetMapping("/report")
	public String  showReport(Map<String, Object> map) {
		String nickNames[] = new String[] {"Raja", "Jani", "King", "RK"};
		Set<Long> mobileNoSet = new HashSet<>();
		mobileNoSet.add(99999999999l);
		mobileNoSet.add(88888888888l);
		List<String> courseList = List.of("Java", "Spring", "Spring  Boot",
"Microservice");
		Map<String, Long> idsMap = Map.of("Aadhar", 34343223l,
                                          "Voter Id", 3435333l,
                                          "Pan No", 553334l);
		//Create model attributes
		map.put("nickNames", nickNames);
		map.put("mobileNoSet", mobileNoSet);
		map.put("courseList", courseList);
		map.put("idsMap", idsMap);
		//retrun LVN
		return "show_report";
	}

}
```

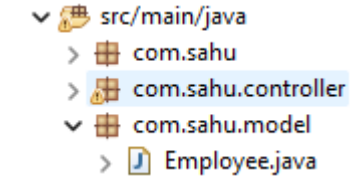
show_report.jsp

```
<%@ page isELIgnored="false" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<h1 style="color: red; text-align: center;">Reading Arrays, Collection
Values</h1>
<b>Nick name : </b><br>
<c:forEach var="name" items="${nickNames}">
	${name}<br>
</c:forEach>
<hr>
```

```
<b>Phone number : </b><br>
<c:forEach var="phNo" items="${mobileNoSet}">
	${phNo}<br>
</c:forEach>
<hr>
<b>Phone number : </b><br>
<c:forEach var="course" items="${courseList}">
	${course}<br>
</c:forEach>
<hr>
<b>Phone number : </b><br>
<c:forEach var="ids" items="${idsMap}">
	${ids.key} : ${ids.value}<br>
</c:forEach>
```

Passing List of Model class objects to View component form Controller class using Data Rendering process

- Create the Employee.java class under com.sahu.model package.



Employee.java

```
package com.sahu.model;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class Employee {
	private int eno;
	private String ename;
	private String desg;
	private double salary;
}
```

TestController.java

```
@Controller
public class TestController {

	@GetMapping("/report")
	public String  showReport(Map<String, Object> map) {
		List<Employee> empList = List.of(
					new Employee(101, "Rajesh", "Clerk",
90000.0),
					new Employee(102, "Mahesh", "Developer",
190000.0),
					new Employee(103, "Anil", "TL", 180000.0)
				);
		map.put("empList", empList);
		//retrun LVN
		return "show_report";
	}

}
```

show_report.jsp

```
<%@ page isELIgnored="false" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<h1 style="color: red; text-align: center;">Reading List<Model> object </h1>
<table border="1" align="center">
	<tr> <th>Employee No</th>
		<th>Employee Name</th>
		<th>Designation</th>
		<th>Salary</th> </tr>
	<c:forEach var="emp" items="${empList}">
		<tr>
			<td>${emp.eno}</td>
			<td>${emp.ename}</td>
			<td>${emp.desg}</td>
			<td>${emp.salary}</td>
		</tr>
	</c:forEach>
</table>
```

Passing Single model class object as model attribute from Controller to View component in Data Rendering process

TestController.java

```
@Controller
public class TestController {

	@GetMapping("/report")
	public String  showReport(Map<String, Object> map) {
		Employee employee = new Employee(101, "Rajesh", "Clerk",
90000.0));
		map.put("employee", employee);
		//retrun LVN
		return "show_report";
	}

}
```

show_report.jsp

```
<%@ page isELIgnored="false" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<h1 style="color: red; text-align: center;">Reading Model object </h1>
<b>Employee No : ${employee. eno}</b><br>
<b >Employee Name : ${employee.ename}</b><br>
<b> Designation : ${employee.desg}</b><br>
<b>Salary : ${employee.salary}</b>
```

Conclusion on Data Rending in Spring Boot MVC application:

- It is the process of passing data from Controller class handler methods to view components through DispatcherServlet using one or another shared memory in the form of Model attributes.
- It can be done in two ways.
  a. Using BindingAwareModelMap object as the shared Memory by taking Map<String, Object> type param in the handler method (Recommended approach)
  b. Using ModelAndView as the return type of handler method (Legacy Approach and not recommended also).

Data Binding

- Passing data from view components to Controller class in the form of Java class object called Model class object.
- It is a process of giving the view components supplied input values to handler methods of controller class.
- View to Controller data passing: Data binding
- Controller to View data passing: Data Rendering
- It can be done in two ways,
    a. Binding form data to Handler method of controller class as the Model/ Command class object using @ModelAttribute (This also called Form binding/ Request wrapping).
    b. Binding hyperlink generated additional request param values to handler method of Controller class using @RequestParam (This also Request param binding).

Form binding/ Request wrapping

- The Java bean class whose object holds form component values of form page is called model class (new name)/ command class (legacy name).
- For Form binding/ Request wrapping/ Data binding using forms we need to the following operations,
    a. Count form components in form page and take same number of properties in Model class.
    b. Make sure that form component names and Model class properties names are matching.
    c. Add getter and setter methods for the properties of Model/ command class.
    d. Take Handler method in Controller class having "@ModelAttribute <Model class> type" parameter.

register.jsp

```
<form action="register_emp" method="POST">
        Name: <input type="text" name="ename"/>
        Address: <input type="text" name="eaddress"/>
        Salary: <input type="text" name="esalary"/>
        <input type="submit" value="register"/>
</form>
```

Employee.java (Model/ Command class)

```
@Data
public class Employee {
```

```
        private String ename;
        private String eaddress;
        private Double salary;
}
```

EmployeeController.java (Controller class)

```
@Controller
public class EmployeeController {

        @RequestMapping(value="/register_emp",
                                        method=RequestMethod.POST)
        (or)
        @PostMapping("/register_emp")
        public String registerEmployee(Map<String,Object> map,
                                @ModelAttribute("emp") Employee emp) {
                ……..
                ……..
        }

}
```

Note: @ModelAttribute is multipurpose annotation. One purpose is to make the parameter of handler method as the Model class and to give instruction to DS to perform Form binding/ Request wrapping operation.

Internal Operations in Form binding/ Request wrapping:

a. End-user fills up form page and submits the request.
b. DS traps and takes the request.
c. DS gets handler method signature through RequestMapping HandlerMapping component.
d. DS notices @ModelAttribute("emp") Employee emp type parameter and understands to perform data binding/ form binding/ request wrapping by taking Employee class as Model class.
e. Creates Model class object having the name given in the @ModelAttribute("emp") as the object name (emp).
   Employee emp = new Employee ();

Note: If @ModelAttribute is taken without param then it takes Model class name like Employee and creates object having class name first letter in lower case as the object name (E.g., employee).

f. Reads the form data by using req.getParmaeter(-) methods and performs necessary conversions like converting received String salary into double salary according data types of Model class properties using PropertyEditor.

```
String name=req.getParameter("ename");
String addrs=req.getParameter("eaddress");
Double salary= Double.parseDouble(req.getParameter("esalary");
```

g. Writes the received and converted form data to Model class object by calling setter method.

```
emp.setEname(name);
emp.setEadd(addrs);
emp.setEsalary(salary);
```

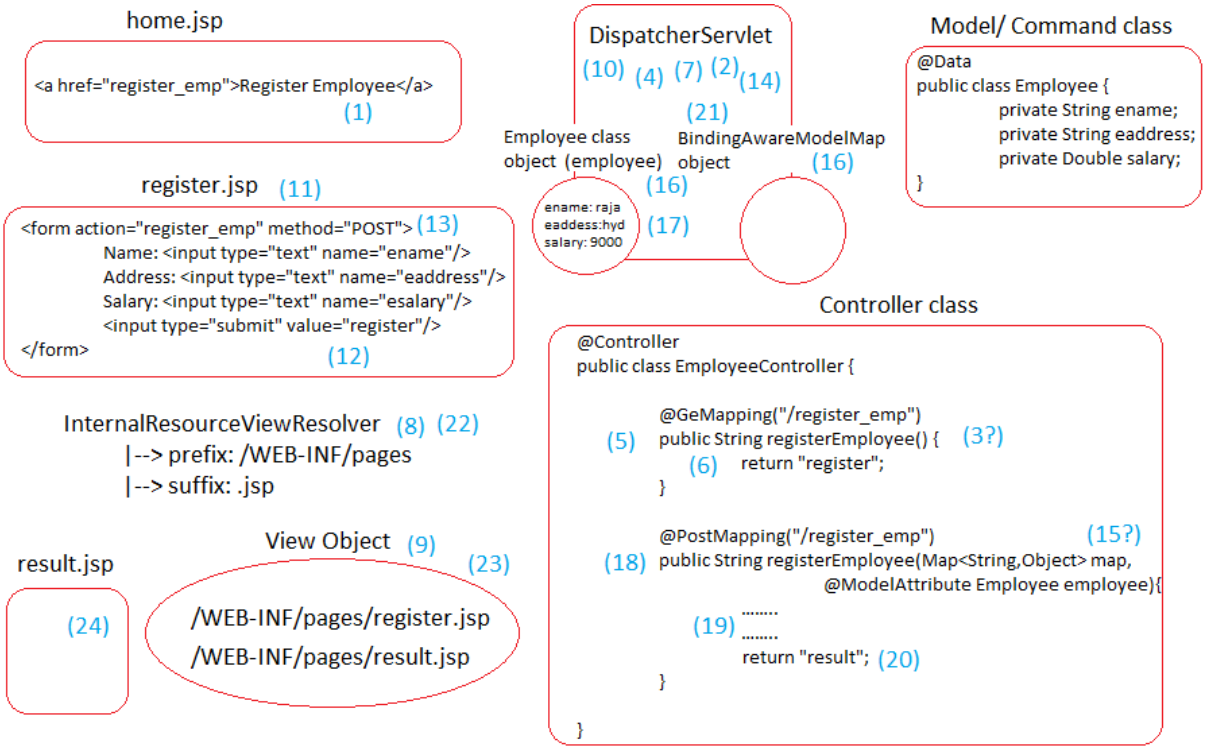
h. DS creates other necessary objects like BindingAwareModelMap object and calls handler method having those objects and model class object (emp).

- Generally, we take two handler methods in controller class with respect form page operation,
  a. Handler method1 in GET mode to launch form page (Showing form page on the browser)
  b. Handler method2 in POST mode to process the form submission request

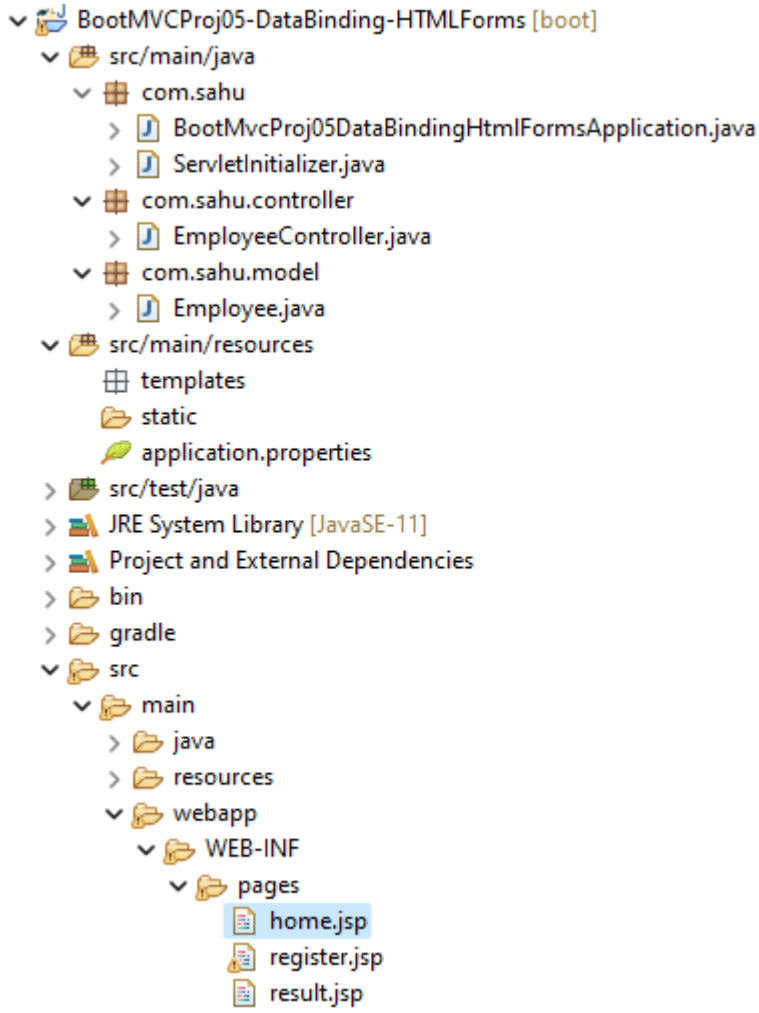
Note:

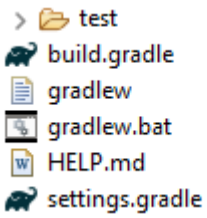
- ✓ Both handler methods can have same request path with different request modes (Recommended).
- ✓ If the above two handler methods are having two different requests paths, then the request modes are your choice.

- Taking same request path for both handler methods (form launching handler method and form submission processing handler method) is going two advantages,
  - Taking "action" attribute in <form> becomes optional.
  - When form page is launched, we can take initial data from Model class object properties and we can display them in form page component as initial values.
    (Possible only when the form page is designed by using Spring supplied JSP tags)



Directory Structure of BootMVCProj05-DataBinding-HTMLForms:





- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also
- Use same Spring Web and Lombok starter during project creation.
- Copy the Tomcat embedded jasper jar (for embedded tomcat) and application.properties from previous project.
- Then place the following code with in their respective files.

Employee.java

```
package com.sahu.model;

import lombok.Data;

@Data
public class Employee {
	private int eno;
	private String ename;
	private String eaddress;
	private Float salary;
}
```

EmployeeController.java

```
package com.sahu.controller;

import java.util.Map;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

import com.sahu.model.Employee;

@Controller
public class EmployeeController {
```

```
	@GetMapping("/")
	public String showHome() {
		return "home";
	}

	@GetMapping("/emp_register")
	public String showForm() {
		return "register";
	}

	@PostMapping("/emp_register")
	public String registerEmployee(Map<String, Object> map,
@ModelAttribute("emp") Employee emp) {
		//Read and use form data from model class object or send to
service class
		return "result";
	}

}
```

home.jsp

```
<h1 style="text-align: center;">
	<a href="emp_register">Register Employee</a>
</h1>
```

register.jsp

```
<%@ page isELIgnored="false"%>

<h1 style="color:green; text-align: center;">Employee Registration
Form</h1>
<form method="post">
	<table align="center">
		<tr>
			<td>Employee No: </td>
			<td><input type="text" name="eno"></td>
		</tr>
		<tr>
			<td>Employee Name: </td>
			<td><input type="text" name="ename"></td>
```

```
            </tr>
            <tr>
                <td>Employee Address: </td>
                <td><input type="text" name="eaddress"></td>
            </tr>
            <tr>
                <td>Employee Salary: </td>
                <td><input type="text" name="salary"></td>
            </tr>
            <tr>
                <td colspan="2">
                    <input type="submit" value="Register">
                </td>
            </tr>
        </table>
</form>
```

result.jsp

```
<%@ page isELIgnored="false"%>

<h1 style="color: red; text-align: center;">Result Page</h1>
<b>Form data : ${emp}</b>
<br>
<a href="./">Home</a>
```

```
@PostMapping("/emp_register")
public String registerEmployee(Map<String, Object> map,
                               @ModelAttribute Employee emp) {
    //Read & use form data from model class object or send to service class
    return "result";
}
```

Here the model class object will be created having class name first letter in lower case as the object name (employee) and will kept in request scope having object name as the request attribute name.

```
Employee employee = new Employee ();
……
…… //data binding (writing form to model class object)
request. setAttribute("employee ", employee);
```

```
@PostMapping("/emp_register")
public String registerEmployee(Map<String, Object> map,
                               @ModelAttribute("emp") Employee emp) {
    //Read & use form data from model class object or send to service class
    return "result";
}
```

Here the model class object will be created having given "emp" name and will also be kept in request scope having "emp" as request attribute name.

```
Employee emp = new Employee ();
……
…… //data binding (writing form to model class object)
request. setAttribute("emp ", employee);
```

JSP tag Libraries

- Spring MVC/ Spring Boot MVC is giving two JSP tag Libraries
  a. Generic tag library (General purpose tags given)
  b. Form tag library (Form design tags are given)

- Every JSP tag library is identified with its taglib URI (fixed) and we need to import that JSP tag library in JSP page by specifying its taglib URI in <%@taglib %> tag in order to use the tags of JSP tag library in JSP page.

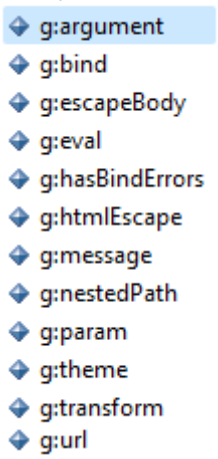
Generic tag library URI

URI: *http://www.springframework.org/tags*

Syntax:

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

The tags are,



Form tag library URI

URI: *http://www.springframework.org/tags/form*

Syntax:

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

The tags are,

- f:button
- f:checkbox
- f:checkboxes
- f:errors
- f:form
- f:hidden
- f:input
- f:label
- f:option
- f:options
- f:password
- f:radiobutton
- f:radiobuttons
- f:select
- f:textarea

- Traditional HTML form tags used in Spring MVC application supports one way data binding i.e., form to model class object.
- Spring MVC supplied JSP form tags used in Spring MVC application supports two-way data binding i.e., form to model class object and model class object to form components.

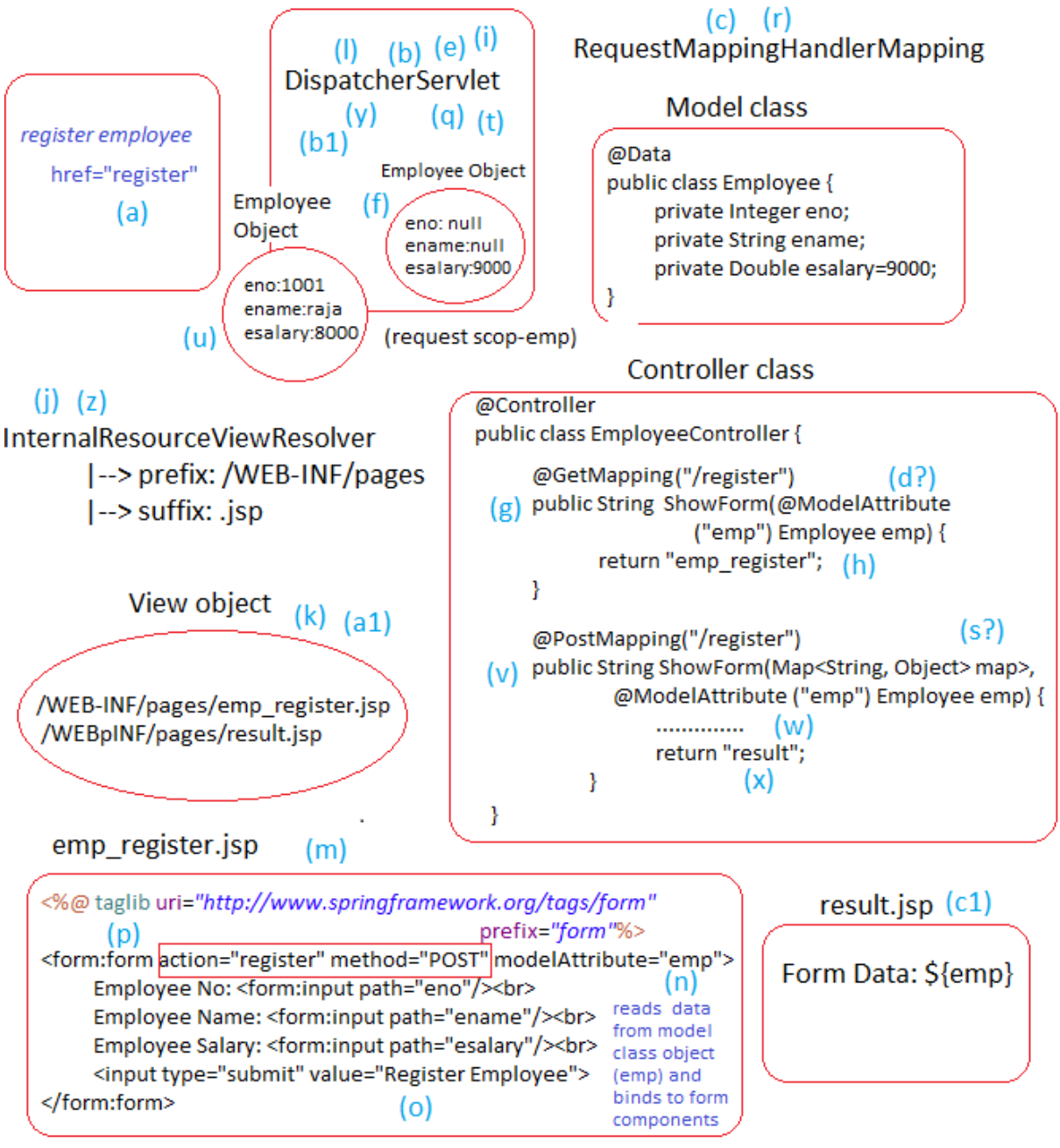
Q. What is the difference between traditional HTML tags and Spring MVC JSP tags?

Ans.

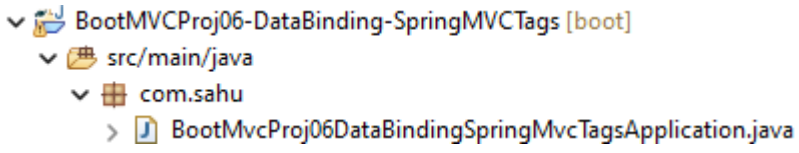
Traditional HTML Form tags	Spring MVC Form tags
a. Supports one way biding (form to model class object).	a. Supports two-way binding (form to model class object and model class object to form).
b. Given by w3c.	b. Given by pivotal team.
c. Default request method for the form generated request is "GET".	c. Default request method for the form generated request is "POST".
d. These tags executed by HTML interpreter.	d. These JSP tags will be converted into html tags having the values collected from model class object as the initial values.

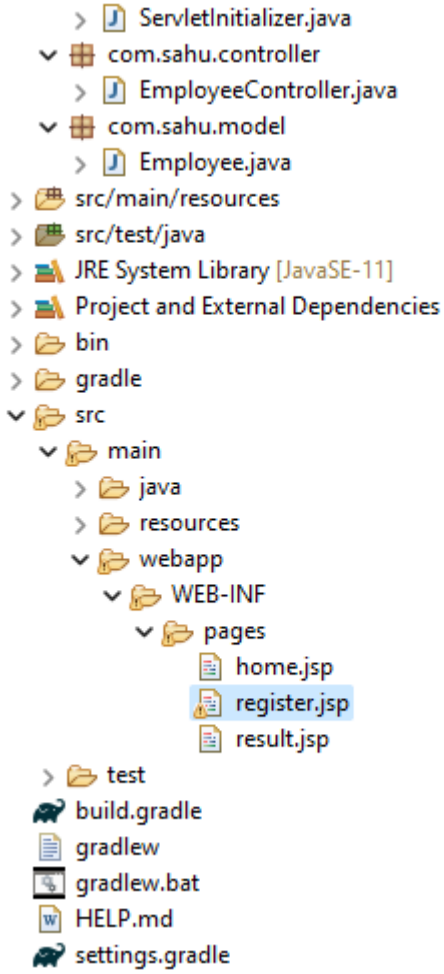
e. Not recommended to use in Spring MVC/ Spring Boot MVC applications.	e. Recommended to use.

Form Binding/ Data Binding/ Request wrapping using Spring MVC supplied Form tags



Directory Structure of BootMVCProj06-DataBinding-SpringMVCTags:





- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also
- Use same Spring Web and Lombok starter during project creation.
- Copy the Tomcat embedded jasper and JSTL jar dependencies and application.properties, home.jsp, result.jsp, Employee.java from previous project.
- Then place the following code with in their respective files.

register.jsp

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

<h1 style="color:green; text-align: center;">Employee Registration
Form</h1>
<form:form modelAttribute="emp">
        <table align="center">
                <tr>
```

```
                <td>Employee No: </td>
                <td><form:input path="eno"/></td>
            </tr>
            <tr>
                <td>Employee Name: </td>
                <td><form:input path="ename"/></td>
            </tr>
            <tr>
                <td>Employee Address: </td>
                <td><form:input path="eaddress"/></td>
            </tr>
            <tr>
                <td>Employee Salary: </td>
                <td><form:input path="salary"/></td>
            </tr>
            <tr>
                <td colspan="2">
                    <input type="submit" value="Register">
                </td>
            </tr>
        </table>
</form:form>
```

EmployeeController.java

```
package com.sahu.controller;

import java.util.Map;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

import com.sahu.model.Employee;

@Controller
public class EmployeeController {

    @GetMapping("/")
    public String showHome() {
```

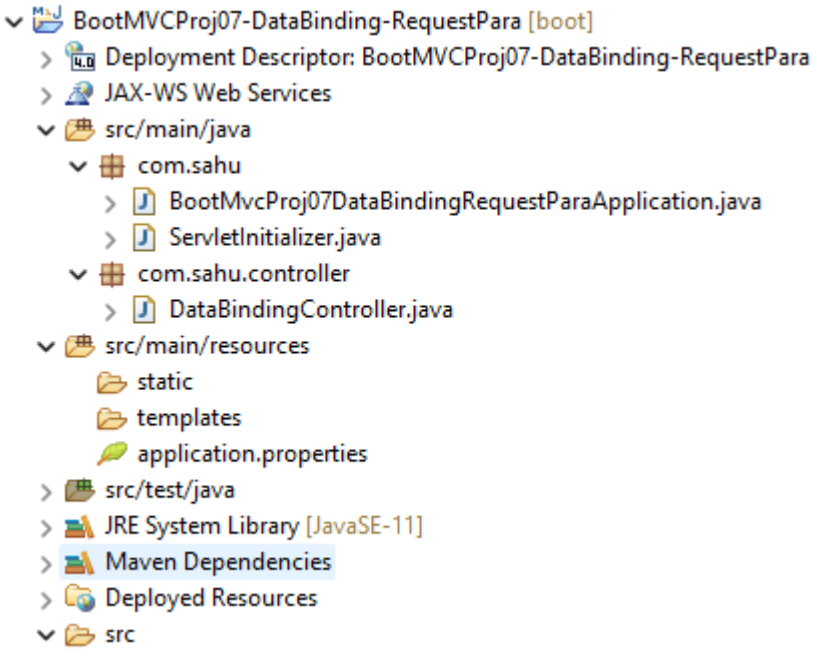
```
                return "home";
        }

        @GetMapping("/emp_register")
        public String showForm(@ModelAttribute("emp") Employee emp) {
                return "register";
        }

        @PostMapping("/emp_register")
        public String registerEmployee(Map<String, Object> map,
@ModelAttribute("emp") Employee emp) {
                //Read and use form data from model class object or send to
service class
                return "result";
        }

}
```

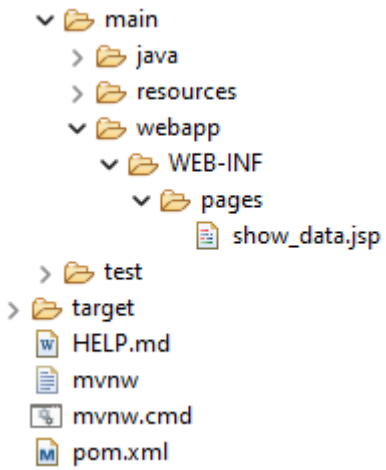
Data Binding using @RequestParam

(Binding request params to handler method params using @RequestParam)

a. The request param in the query String either directly using hyperlink can be bound with handler method params of controller class using the support of @RequestParam annotation.

Directory Structure of BootMVCProj07-DataBinding-RequestPara:





- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also
- Use same Spring Web and Lombok starter during project creation.
- Copy the Tomcat embedded jasper jar (for embedded tomcat), and application.properties from previous project.
- Then place the following code with in their respective files.

DataBindingController.java

```
package com.sahu.controller;

import java.util.Map;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class DataBindingController {

	@GetMapping("/data")
	public String bindData(Map<String, Object> map,
				@RequestParam("sno") Integer no,
				@RequestParam("sname") String name) {
		System.out.println(no+" "+name);
		//return LVN
		return "show_data";
	}

}
```

show_data.java

```
<%@ page isELIgnored="false"%>

<h1 style="text-align: center; color: red;">Result Page</h1>
<b>SNo: ${param.sno}</b><br>
<b>SName: ${param.sname}</b>
```

Request: http://localhost:2525/BootMVCProj07-DataBinding-RequestPara/data?sno=1001&sname=raj

b. If the request param specified @RequestParam annotation is given not given then 400 bad request error will come.
WARN 13840 --- [nio-2525-exec-5] .w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.web.bind.MissingServletRequestParameterException: Required request parameter 'sname' for method parameter type String is not present]

DataBindingController.java

```
@Controller
public class DataBindingController {

	@GetMapping("/data")
	public String bindData(Map<String, Object> map,
				@RequestParam Integer sno,
				@RequestParam String name) {
		System.out.println(no+" "+name);
		//return LVN
		return "show_data";
	}

}
```

Request: http://localhost:2525/BootMVCProj07-DataBinding-RequestPara/data?sno=1001&sname=raj

Solution: Either give "name" request param in the URL, and make the method parameter name is “sname” or make the "name" request param presence in the request URL optional as shown below.

DataBindingController.java

```
@Controller
public class DataBindingController {

	@GetMapping("/data")
	public String bindData(Map<String, Object> map,
				@RequestParam Integer no,
				@RequestParam(required = false) String name) {
		System.out.println(no+" "+name);
		//return LVN
		return "show_data";
	}

}
```

c. If do not give value for the request param in the request URL then it will take " "(empty string) as the default value but we can override that default value with our choice default value using "defaultValue" param of @RequestParam annotation.

DataBindingController.java

```
@Controller
public class DataBindingController {

	@GetMapping("/data")
	public String bindData(Map<String, Object> map,
			@RequestParam Integer sno,
			@RequestParam(defaultValue = "raja") String sname) {
		System.out.println(sno+" "+sname);
		//return LVN
		return "show_data";
	}

}
```

Request: http://localhost:2525/BootMVCProj07-DataBinding-RequestPara/data?sno=1001
(or)
http://localhost:2525/BootMVCProj07-DataBinding-RequestPara/data?sno=1001&sname=

d. If the handler method parameter type is "non-string" type and your giving string values in the request param values of request URL then you will get NumberFormatException.
WARN 4160 --- [nio-2525-exec-9]
.w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.web.method.annotation.MethodArgumentTypeMismatchException: Failed to convert value of type 'java.lang.String' to required type 'java.lang.Integer'; nested exception is java.lang.NumberFormatException: For input string: "101f"]

Request: http://localhost:2525/BootMVCProj07-DataBinding-RequestPara/data?sno=101f&sname=anil

e. If handler method parameter type is non-string type and we are passing no value or " " for the relevant request parameter in the request URL then we get "MissingServletRequestParameterException" as show below.
WARN 4160 --- [io-2525-exec-10]
.w.s.m.s.DefaultHandlerExceptionResolver : Resolved [org.springframework.web.bind.MissingServletRequestParameterException: Required request parameter 'sno' for method parameter type Integer is present but converted to null]

Request: http://localhost:2525/BootMVCProj07-DataBinding-RequestPara/data?sno=&sname=anil

Solution 1: Specify the default value.
DataBindingController.java

```
@Controller
public class DataBindingController {

	@GetMapping("/data")
	public String bindData(Map<String, Object> map,
			@RequestParam(defaultValue = "1001")  Integer sno,
			@RequestParam(defaultValue = "raja") String sname) {
		System.out.println(sno+" "+sname);
		//return LVN
		return "show_data";
	}

}
```

Solution 2: Take the type as wrapper type and make it optional request param.

DataBindingController.java

```
@Controller
public class DataBindingController {

	@GetMapping("/data")
	public String bindData(Map<String, Object> map,
			@RequestParam(required = false)  Integer sno,
			@RequestParam(defaultValue = "raja") String sname) {
		System.out.println(sno+" "+sname);
		//return LVN
		return "show_data";
	}

}
```

f. If one request param is having multiple values then we can store them either in String [] type or List/ Set collection type parameter of handler method (same thing is applicable while binding form data to Model class object).

DataBindingController.java

```
@Controller
public class DataBindingController {

	@GetMapping("/data")
	public String bindData(Map<String, Object> map,
			@RequestParam Integer sno,
			@RequestParam String[] sname,
			@RequestParam("sname") List<String> names) {
		System.out.println(sno+" "+sname);
		//return LVN
		return "show_data";
	}

}
```

Request: http://localhost:2525/BootMVCProj07-DataBinding-RequestPara/data?sno101=&sname=anil&sname=raja

g. If the Handler method @RequestParam annotation-based request param type is "String" but we give multiple values for that request parameter then it will store those multiple values to handler method param as comma separated single string of values.

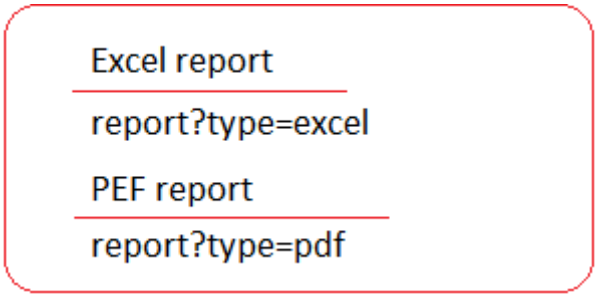
DataBindingController.java

```
@Controller
public class DataBindingController {

	@GetMapping("/data")
	public String bindData(Map<String, Object> map,
			@RequestParam Integer sno,
			@RequestParam String sname ){
		System.out.println(sno+" "+sname);
		//return LVN
		return "show_data";
	}

}
```

Request: http://localhost:2525/BootMVCProj07-DataBinding-RequestPara/data?sno101=&sname=anil&sname=raja

Note: If form page is having 1 or 2 form components then we can bind them directly to handler methods params using @RequestParam annotation rather taking separate Model class.

@RequestParam Use cases:
Case 1: To differentiate logics for the hyperlink generated requests



Controller class

```
@Controller
public class ReportController {
```

```
        @GetMapping("/report")
        public String generateReport(Map<String,Object> map,
                                                    @RequestParam String type) {
                if (type.equalsIgnoreCase("excel") {
                        …… // talk to service class to report data
                }
                else {
                        …… // talk to service class to report data
                }
                //return LVN
        }
}
```

Case 2:

Web Page

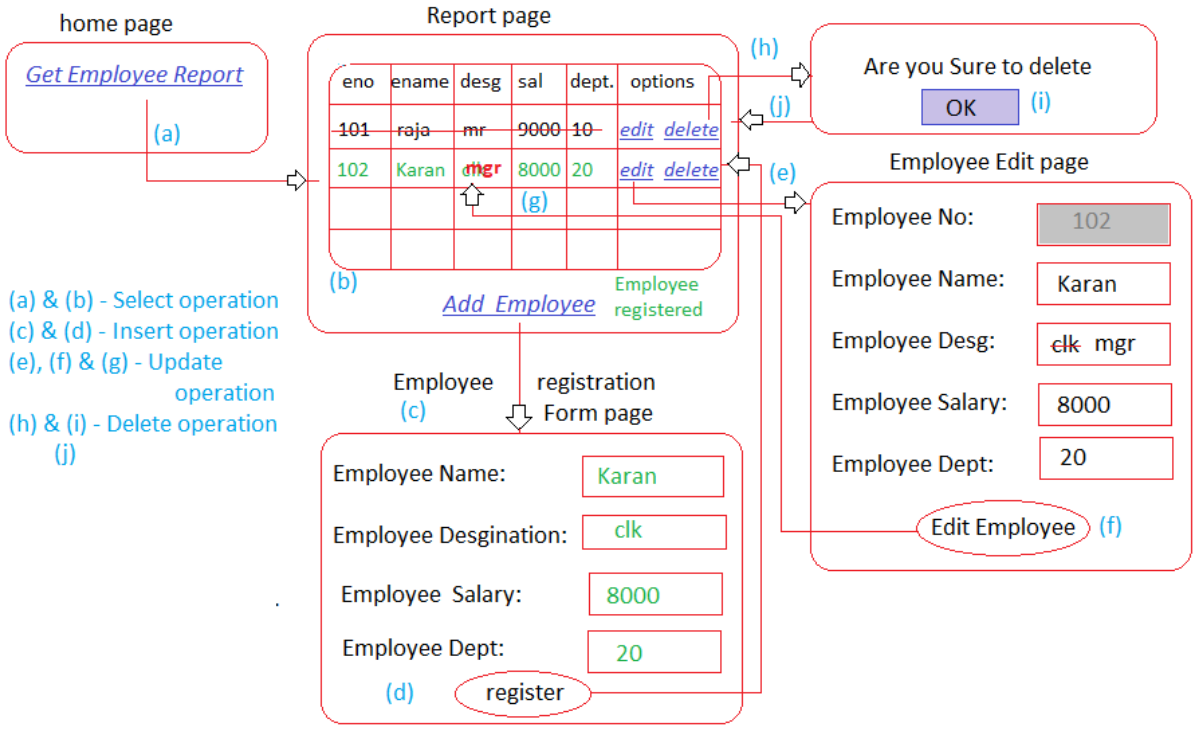
Sales Report

pid	pname	price	opt1	opt2
101	table	9000	*edit* edit?pid=101	*delete* delete?pid=101
102	chair	8000	*edit* edit?pid=102	*delete* delete?pid=102

Controller class

```
@Controller
public class ProductController {
        @GetMapping("/edit')
        public String editProduct(Map<String,Object>,
                                                    @RequestParam int pid){
                ……..
        }
        @GetMapping("/delete"
        public String deleteProduct(Map<String,Object>,
                                                    @RequestParam int pid){
                ……
        }
}
```

Spring Boot MVC with Spring Data JPA performing CURD operations



Hard deletion: The record will be deleted from DB table physically.

E.g., Ticket cancellation

Soft deletion: The record will not be deleted physically but it will be marked as deleted in status column. It does not participate any persistence operations.

E.g., Bank account closing

- Server Managed JDBC connection pool taken in Spring MVC app can be used in External servers' deployment but cannot used in Embedded servers

  E.g., Tomcat server managed JDBC connection pool, Wildfly server managed JDBC connection pool
- Third party JDBC connection pool like HikariCP taken in Spring MVC app can be used in any both Embedded server and External server.

  E.g., HikariCP (best), Apache DBCP2, TomcatCP, C3P0, ViburCP, Proxool and etc.

Steps for Developing Spring boot MVC Mini Project

Step 1: Create Spring Boot starter project by adding the following staters.

- [X] Lombok
- [X] Spring Data JPA
- [X] Oracle Driver
- [X] Spring Web

Step 2: Create application.properties file as show below.

application.properties

```
#ViewResolver
spring.mvc.view.prefix=/WEB-INF/pages/
spring.mvc.view.suffix=.jsp

#Embedded Server port number
server.port=4547

#Context path for standalone execution
server.servlet.context-path=/Employee-CURD

#Connection provider to work HikariCP
spring.datasource.driver-class-name==oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager
spring.datasource.hikari.minimum-idle=10
spring.datasource.hikari.maximum-pool-size=100
spring.datasource.hikari.idle-timeout=60000

#Spring Data JPA Hibernate properties
spring.jpa.database-platform=org.hibernate.dialect.Oracle10gDialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
```

Step 3: Develop Model class or entity class.
Step 4: Develop repository interface.
Step 5: Write code in controller class to lunch the home page.
Step 6: Develop home.jsp in WEB-INF/pages folder.
Step 7: Develop Service interface and service implementation class with respect to report generation.
Step 8: Develop the controller support report generation.
Step9: Develop employee_report.jsp in WEB-INF/pages folder using JSTL tags to display employees' details in the form html table (Add JSTL Dependency in pom.xml).
Step 10: Develop the entire application by following code.
Step 11: Then run the application.

Before that we will read all required things then we will develop application.

Note:

- ✓ Every JSP tag library is identified with its taglib URI and we need import JSP tag library in our JSP page by specifying its taglib URI (fixed) and our choice prefix with the support of <%@taglib %> Directive tag.
- ✓ JSTL tags are given to make our JSP pages as Java code less JSP pages i.e., either we can avoid Java code or we can minimize Java code from the JSP pages (view Helper Pattern).
- ✓ To support View Helper pattern in JSP pages we need to do the following things,
    - Avoid or minimize the utilization of scripting tags (scriptlet, expression, declaration).
    - Use html tags, JSP directive tags, standard action tags, JSP EL, JSTL tags, Custom JSP tags.

Edit Employee Operation:

- ➢ Here in form launch/ initial phase we need to display form page having selected employee details as the initial values and providing ability to modify the values in text boxes except PK column value, when form is submitted (post back phase) the record should be updated with new value based on pk value as the criteria value.

Q. What is double posting problem and how can solve this problem in Spring MVC/ Spring Boot MVC?

Ans.

- Executing form submission logic for multiple times by pressing refresh button on the result page of form submission or submitting form by getting it through back button is called double posting or duplicate form submission problem.
- This scenario gives problem when the POST mode request given by form page is performing some non-select operation on application's data (repetition of business logic by generating request through refresh button).

Side effects of Double Posting problem:

a. Taking online payment for multiple times.
b. Inserting duplicate information with different PK column values in the registration process
and etc.

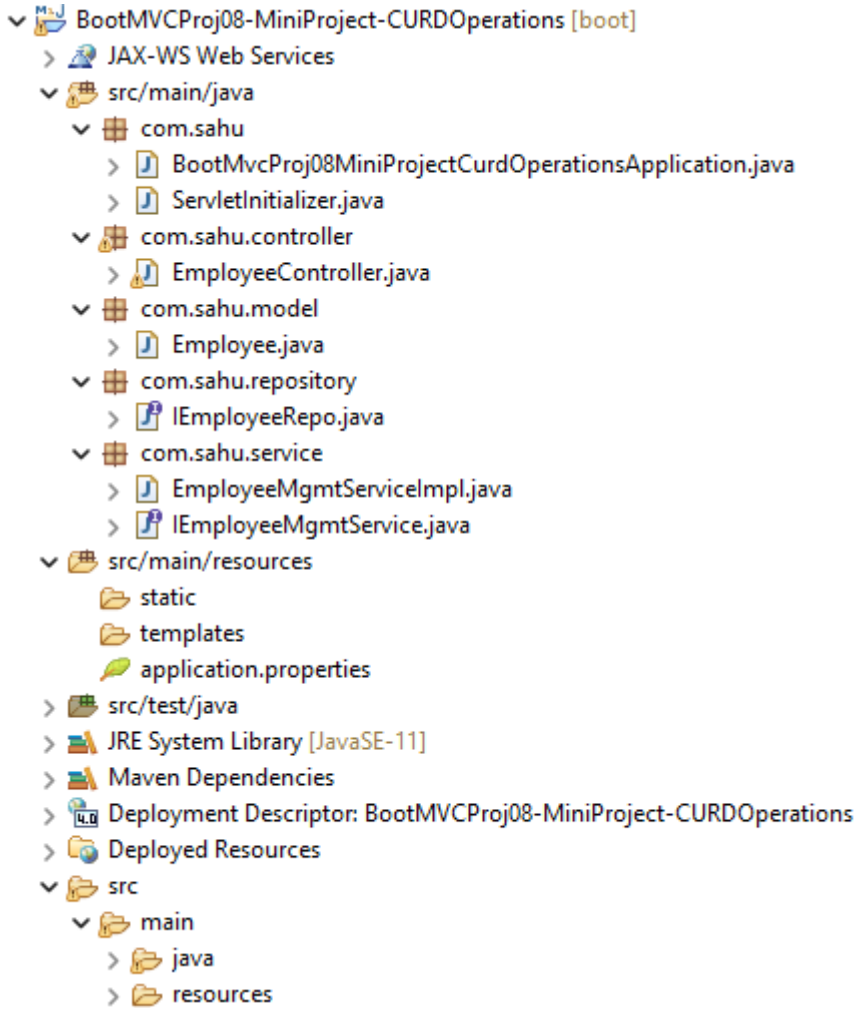
- ➢ Solution for this problem in Servlet, JSP environment is Session tokens.

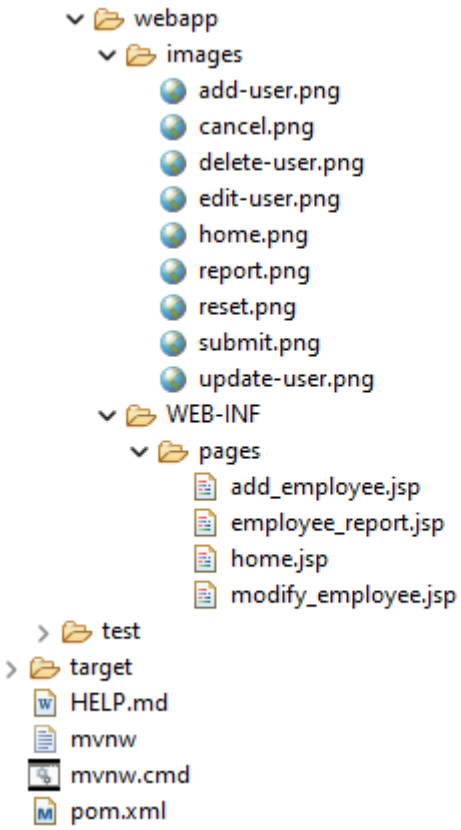
- Solution for this problem in Spring or Spring MVC environment is implementing PRG pattern.

PRG Pattern (Post/Redirect/Get Pattern):

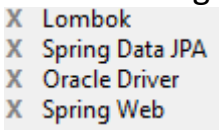
- This pattern says do not launch result page using a handler method that handles POST mode request, redirect that request another handler that method that handles GET mode request and show result page from that method.
- Due to this if repeat the request using refresh button, the logic that is placed in handler method handling "GET" mode request executes.
- Generally, "GET" mode requests perform select operations (read only) in the web application, so repetition of these logics does not give any problem,
- Generally, "POST" requests perform non-select operations (update, insert, delete) in the web application, so repetition of these logics definitely gives problem.

Directory Structure of BootMVCProj08-MiniProject-CURDOperations:





- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also
- Use following starters during project creation.



- Add the Tomcat embedded jasper jar (for embedded tomcat), JSTL jar dependency from MVN repository in pom.xml.
- Collect the images from internet.
- Then place the following code with in their respective files.

application.properties

```
#ViewResolver
spring.mvc.view.prefix=/WEB-INF/pages/
spring.mvc.view.suffix=.jsp

#Embedded Server port number
server.port=4547

#Context path for standalone execution
server.servlet.context-path=/Employee-CURD
```

```
#Connection provider to work HikariCP
spring.datasource.driver-class-name==oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager
spring.datasource.hikari.minimum-idle=10
spring.datasource.hikari.maximum-pool-size=100
spring.datasource.hikari.idle-timeout=60000

#Spring Data JPA Hibernate properties
spring.jpa.database-platform=org.hibernate.dialect.Oracle10gDialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update
```

Employee.java

```
package com.sahu.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;

import lombok.Data;

@Entity
@Table(name="emp")
@Data
public class Employee {
	@Id
	@SequenceGenerator(name = "gen1", sequenceName =
"emp_no_seq1", initialValue = 3000, allocationSize = 1)
	@GeneratedValue(generator = "gen1", strategy =
GenerationType.SEQUENCE)
	private Integer empno;
	private String ename;
	private String job;
	private Float sal;
	private Integer deptno;
}
```

IEmployeeRepo.java

```
package com.sahu.repository;

import org.springframework.data.repository.PagingAndSortingRepository;

import com.sahu.model.Employee;

public interface IEmployeeRepo extends
PagingAndSortingRepository<Employee, Integer> {

}
```

IEmployeeMgmtService.java

```
package com.sahu.service;

import com.sahu.model.Employee;

public interface IEmployeeMgmtService {
	public Iterable<Employee> getAllEmployees();
	public String insertEmployee(Employee emp);
	public Employee getEmployeeByEmpNo(Integer empno);
	public String updateEmployee(Employee emp);
	public String deleteEmployeeByEno(Integer eno);
}
```

EmployeeMgmtServiceImpl.java

```
package com.sahu.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.data.domain.Sort;
import org.springframework.stereotype.Service;

import com.sahu.model.Employee;
import com.sahu.repository.IEmployeeRepo;

@Service("empService")
public class EmployeeMgmtServiceImpl implements
IEmployeeMgmtService {

```

```
		@Autowired
	private IEmployeeRepo employeeRepo;

	@Override
	public Iterable<Employee> getAllEmployees() {
		return employeeRepo.findAll(Sort.by("job").ascending());
	}

	@Override
	public String insertEmployee(Employee emp) {
		return emp.getEname()+" details has registered and his
Employee id is "+employeeRepo.save(emp).getEmpno();
	}

	@Override
	public Employee getEmployeeByEmpNo(Integer empno) {
		return employeeRepo.findById(empno).get();
	}

	@Override
	public String updateEmployee(Employee emp) {
		return employeeRepo.save(emp).getEname()+" details has
updated";
	}

	@Override
	public String deleteEmployeeByEno(Integer eno) {
		employeeRepo.deleteById(eno);
		return "Employee Id - "+eno+" details has deleted";
	}

}
```

EmployeeController.java

```
package com.sahu.controller;

import java.util.Map;

import javax.servlet.http.HttpSession;
```

```java
import org.springframework.beans.BeanUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.mvc.support.RedirectAttributes;

import com.sahu.model.Employee;
import com.sahu.service.IEmployeeMgmtService;

@Controller
public class EmployeeController {

	@Autowired
	private IEmployeeMgmtService employeeMgmtService;

	@GetMapping("/")
	public String showHomePage() {
		return "home";
	}

	@GetMapping("/emp_report")
	public String showEmployeeReport(Map<String, Object> map) {
		//User Service
		Iterable<Employee> empsList =
employeeMgmtService.getAllEmployees();
		//Keep results as model attribute
		map.put("empsList", empsList);
		//return LVN
		return "employee_report";
	}

	@GetMapping("/add_employee")
	public String showAddEmployeeForm(@ModelAttribute("emp")
Employee emp) {
		return "add_employee";
	}
```

```
        @PostMapping("/add_employee")
        public String insertEmployee(HttpSession session,
@ModelAttribute("emp") Employee emp) {
                //Use service
                String result = employeeMgmtService.insertEmployee(emp);
                //Add result to HttpSession object
                session.setAttribute("resultMsg", result);
                //return LVN
                return "redirect:emp_report";
        }

        @GetMapping("/edit_employee")
        public String showEditEmployeeFormPage(@RequestParam("eno")
Integer eno,
                                               @ModelAttribute("emp")
Employee emp) {
                //use service
                Employee employee =
employeeMgmtService.getEmployeeByEmpNo(eno);
                BeanUtils.copyProperties(employee, emp);
                //return LVN
                return "modify_employee";
        }

        @PostMapping("/edit_employee")
        public String editEmployee(RedirectAttributes attrs,
@ModelAttribute("emp") Employee emp) {
                //Use service
                String result = employeeMgmtService.updateEmployee(emp);
                //Add result to RedirectAttributes object
                attrs.addFlashAttribute("resultMsg", result);
                //return LVN
                return "redirect:emp_report";
        }

        @GetMapping("/delete_employee")
        public String deleteEmployee(@RequestParam("eno") Integer eno,
                                                RedirectAttributes attrs) {
                //use service
                String result =
```

```
employeeMgmtService.deleteEmployeeByEno(eno);
                //Add result to RedirectAttributes object
                attrs.addFlashAttribute("resultMsg", result);
                //return LVN
                return "redirect:emp_report";
        }

}
```

home.jsp

```
<%@ page isELIgnored="false"%>

<h1 style="text-align: center;">
        <a href="emp_report">
                <img src="images/report.png"/>
        </a>
</h1>
```

employee_report.jsp

```
<%@ page isELIgnored="false"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<c:choose>
        <c:when test="${!empty empsList}">
                <table border="1" bgcolor="cyan" align="center">
                        <tr bgcolor="pink">
                                <th>ENo</th>
                                <th>EName</th>
                                <th>Desg</th>
                                <th>Salary</th>
                                <th>Dept No</th>
                                <th>Operations</th>
                        </tr>
                        <c:forEach var="emp" items="${empsList}">
                                <tr>
                                        <td>${emp.empno}</td>
                                        <td>${emp.ename}</td>
                                        <td>${emp.job}</td>
                                        <td>${emp.sal}</td>
```

```
                                <td>${emp.deptno}</td>
                                <td>
                                    <a
href="edit_employee?eno=${emp.empno}">
                                        <img class="user-operation"
src="images/edit-user.png" />
                                    </a>   
                                    <a
href="delete_employee?eno=${emp.empno}" onclick="confirm('Do you
want to delete?')">
                                        <img class="user-operation"
src="images/delete-user.png" />
                                    </a>
                                </td>
                            </tr>
                        </c:forEach>
                    </table>
                </c:when>
                <c:otherwise>
                    <h1 style="color: red; text-align: center;">Records not
Found</h1>
                </c:otherwise>
            </c:choose>
            <br>
                <h1 class="blink_me" style="color: green; text-align:
center;">${resultMsg}</h1>
            <br>
            <div style="text-align: center;">
                <a href="add_employee">
                    <img class="user-operation" src="images/add-user.png" />
                </a>
                  
                <a href="./">
                    <img class="user-operation" src="images/home.png" />
                </a>
            </div>

            <style>
            .user-operation {
                height: 50px;
```

```
        width: 50px;
}

.blink_me {
  animation: blinker 1s linear infinite;
}

@keyframes blinker {
  50% {
    opacity: 0;
  }
}
</style>
```

add_employee.jsp

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

<h1 style="color: blue; text-align: center;">Register Employee</h1>

<form:form modelAttribute="emp">
        <table border="0" bgcolor="cyan" align="center">
                <tr>
                        <td>Employee Name : </td>
                        <td><form:input path="ename"/> </td>
                </tr>
                <tr>
                        <td>Employee Designation : </td>
                        <td><form:input path="job"/> </td>
                </tr>
                <tr>
                        <td>Employee Salary : </td>
                        <td><form:input path="sal"/> </td>
                </tr>
                <tr>
                        <td>Employee Dept No : </td>
                        <td><form:input path="deptno"/> </td>
                </tr>
                <tr>
```

```
                    <td></td>
                    <td>
                        <button type="reset">
                            <img class="operation-btn" alt="rest image"
src="images/reset.png">
                        </button>
                          
                        <input class="operation-btn" type="image"
src="images/submit.png">
                    </td>
                </tr>
            </table>
</form:form>

<style>
.operation-btn{
        width: 50px;
        height: 50px;
}
</style>
```

modify_employee.jsp

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

<h1 style="color: blue; text-align: center;">Edit Employee</h1>

<form:form modelAttribute="emp">
        <table border="0" bgcolor="cyan" align="center">
                <tr>
                        <td>Employee No : </td>
                        <td><form:input path="empno" readonly="true"/></td>
                </tr>
                <tr>
                        <td>Employee Name : </td>
                        <td><form:input path="ename"/> </td>
                </tr>
                <tr>
                        <td>Employee Designation : </td>
```

```
                <td><form:input path="job"/> </td>
            </tr>
            <tr>
                <td>Employee Salary : </td>
                <td><form:input path="sal"/> </td>
            </tr>
            <tr>
                <td>Employee Dept No : </td>
                <td><form:input path="deptno"/> </td>
            </tr>
            <tr>
                <td></td>
                <td>
                    <button type="reset">
                        <img class="operation-btn" alt="rest image"
src="images/cancel.png">
                    </button>
                      
                    <input class="operation-btn" type="image"
src="images/update-user.png">
                </td>
            </tr>
        </table>
</form:form>

<style>
.operation-btn{
        width: 50px;
        height: 50px;
}
</style>
```

Q. What is difference b/w flashAttributes of RedirectAttributes object and Session Attributes?

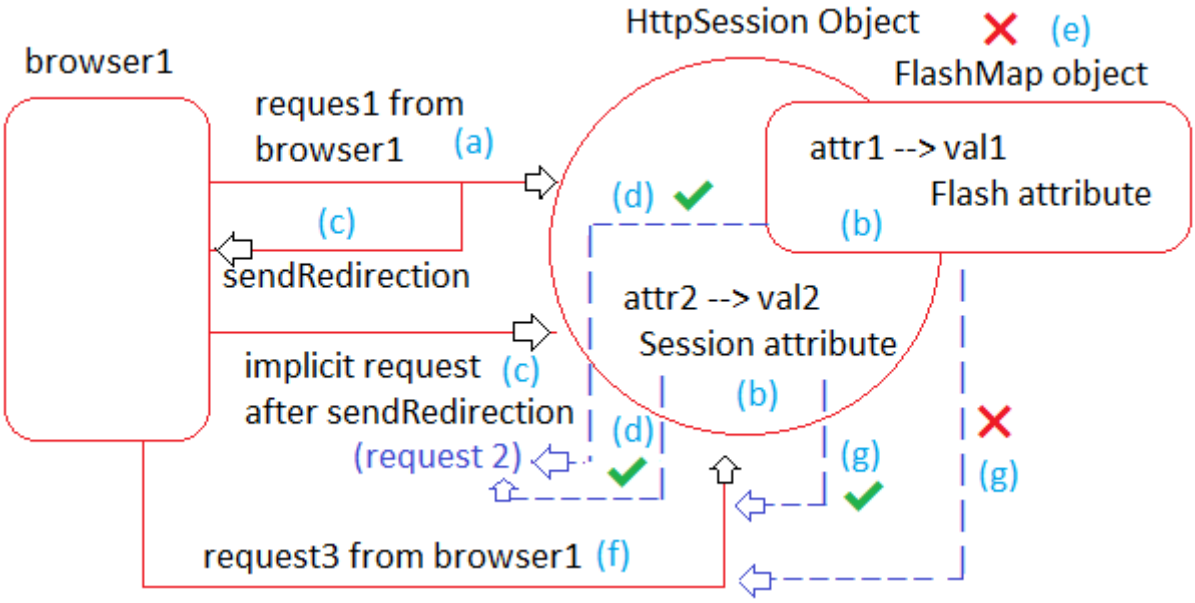
Ans.

- FlashAttributes of RedirectAttributes object allocates Memory in FlashMap object (given by Spring API internally inside Session object) which will created before redirection and will be made available and removed after redirection. So, it is very useful while implementing PRG Pattern.

- A FlashMap provides a way for one request to store attributes intended for use in another. This is most commonly needed when redirecting from one URL to another e.g., the Post/Redirect/Get pattern. A FlashMap is saved before the redirect (typically in the session) and is made available after the redirect and removed immediately.

Note: FlashMap object allocates memory inside the Session object but will be removed after redirection process.

- Session Attributes allocates memory in Session object and will be stayed forever across multiple requests given browser s/w (Client) for whom this Session object and session attributes created



- Flash attributes are visible between two requests of sendRedirection by allocating memory inside session object on temporary basis (a to c).
- Session attributes are visible across the all requests given by browser/ client by allocating memory inside the session object for long time (as long as session continues b/w browser and web application) (a to f).

Q. What is the difference b/w model attributes of BindingAwareModelMap object and flash attribute of RedirectAttributes object?

Ans.

- Model attributes scope is request scopes i.e., they cannot be used across the multiple requests coming from a client (browser) or across the initial and later requests of send redirection. These attributes allocate memory in BindingAwareModelMap object whose scope is request scope.

- Flash attributes scope is redirection scope i.e., they are created before redirection and will be made available after redirection and will be deleted immediately. These attributes allocate memory in FlashMap object of Session object.

Note:

- ✓ Redirect attributes object can create both normal Model attributes and Flash attributes.
- ✓ Model attributes allocates memory inside BindingAwareModelMap object having request scope whereas Flash attributes allocates memory in FlashMap object of Session object.

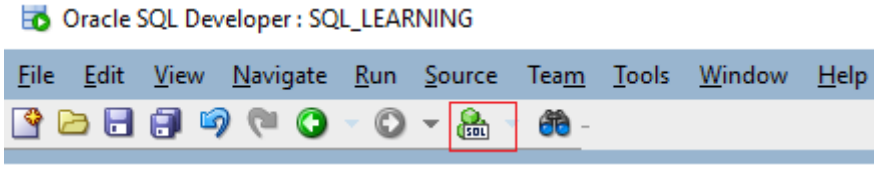
Soft Deletion in Spring MVC and Spring Data JPA

- Deleting records permanently from DB table is called hard deletion.
  E.g., Ticket cancellation, Booking cancellation and etc.
- Marking the record as DELETED record by changing the status of the record and making that not participating in CURD operations is called Soft Deletion.
  E.g., Bank Account closing, Employee resignation and etc.
- We have JPA annotations called SQLXxx annotation which allows to configure custom SQL Queries for standard SQL operations like insert, update, delete.
- @SQLDelete annotation is useful to configure custom query for the standard delete operation. This is veery to configure UPDATE SQL query of soft deletion to mark the record as DELETED record for standard delete operation.
- @Where annotation can be used to specify implicit conditions that should be applied on every query executes.

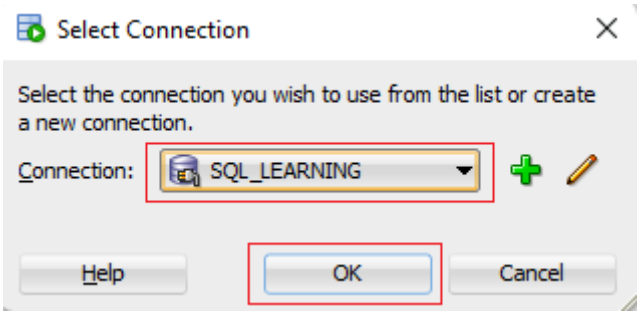
Steps for soft Deletion

a. Create DB table having status column with "active" value for all records.

Step 1: Open the SQL Worksheet.

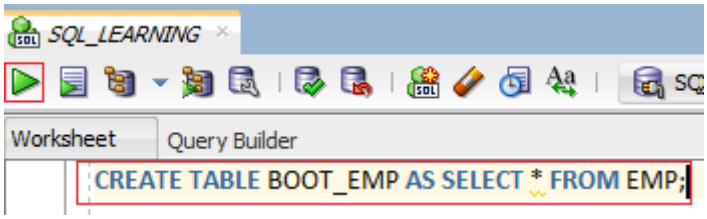


Step 2: Select a Connection and click on OK.

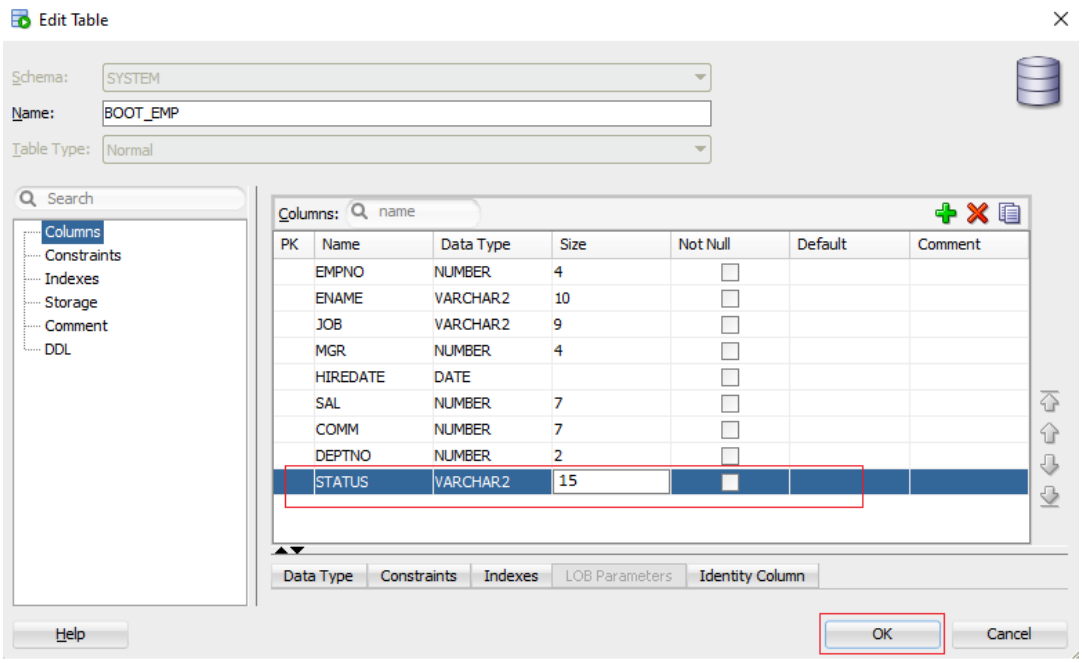


Step 3: Use the below query execute it and create the new table.

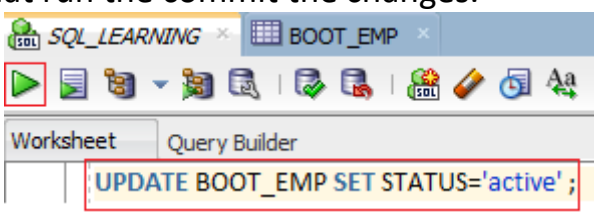
```
CREATE TABLE BOOT_EMP AS SELECT * FROM EMP;
```



Step 4: Add STATUS column to the DB table BOOT_EMP.



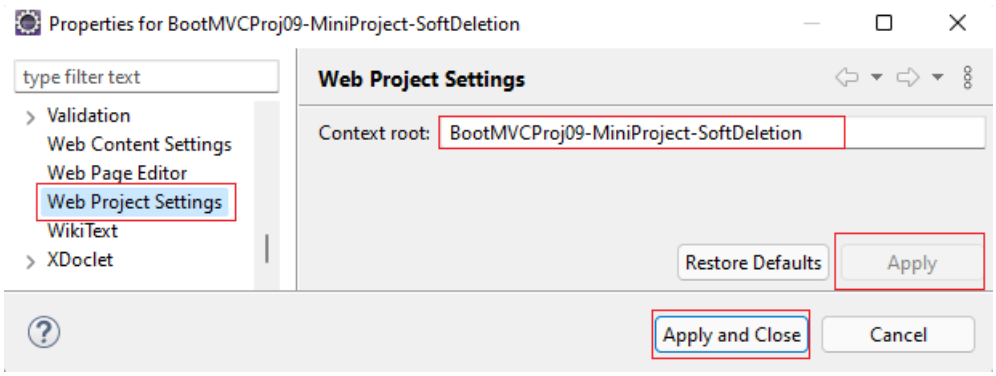
Step 5: Update the STATUS column with ‘active’ by running the below query after that run the commit the changes.



b. Add @SQLDelete annotation specifying update query of soft deletion (write on the top of Entity class).
c. Add @Where annotation specify condition to eliminate softly deleted records from regular CURD operation.

Directory Structure of BootMVCProj09-MiniProject-SoftDeletion:

- Copy and paste the BootMVCProj08-MiniProject-CURDOperations and rename to BootMVCProj09-MiniProject-SoftDeletion.
- After that change the Web Project Setting context root to BootMVCProj09-MiniProject-SoftDeletion.
- For that right click on project go to Properties then, go to Web Project Settings and change the Context root name new project name then click on Apply then Apply and Close.



- Then change in pom.xml also in following places.

```
<groupId>sahu</groupId>
<artifactId>BootMVCProj09-MiniProject-SoftDeletion</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>war</packaging>
<name>BootMVCProj09-MiniProject-SoftDeletion</name>
<description>Mini project CUR operation</description>
```

- Then place the following code with in their respective files.

Employee.java

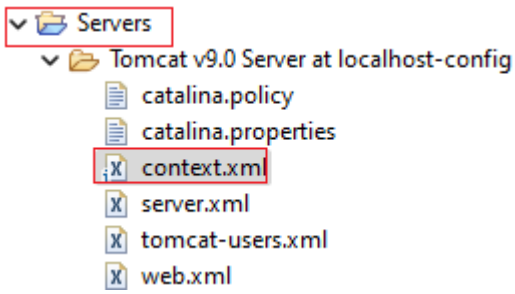
```
@Entity
@Table(name="boot_emp")
@Data
@SQLDelete(sql="UPDATE BOOT_EMP SET STATUS='deleted' WHERE
EMPNO=?")
@Where(clause = "STATUS <>'deleted'")
```

```
public class Employee {
	@Id
	@SequenceGenerator(name = "gen1", sequenceName =
"emp_no_seq1", initialValue = 3000, allocationSize = 1)
	@GeneratedValue(generator = "gen1", strategy =
GenerationType.SEQUENCE)
	private Integer empno;
	private String ename;
	private String job;
	private Float sal;
	private Integer deptno;
	private String status="active";
}
```

- Using Repository methods like delete (-) we can do only soft deletion on this DB table using this Entity class.
- To perform hard deletion on this DB table we can use &Query methods of repository either with JPQL or Native SQL.

Procedure to configure Tomcat server managed JDBC connection pool in Spring Boot MVC application

Step 1: Create JDBC DataSource with JDBC connection pool for oracle in Tomcat server by adding <Resource> entries under tag in context.xml file of "servers" section from Eclipse Project Explorer [Click here to collect the Information].




```
<Resource name="jdbc/myoracle" auth="Container"
	type="javax.sql.DataSource"
	driverClassName="oracle.jdbc.OracleDriver"
	url="jdbc:oracle:thin:@127.0.0.1:1521:mysid"
	username="scott" password="tiger" maxTotal="20" maxIdle="10"
	maxWaitMillis="-1"/>
```

context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
  Licensed to the Apache Software Foundation (ASF) under one or

<Context>
    <Resource name="DsJndi" auth="Container"
        type="javax.sql.DataSource"
        driverClassName="oracle.jdbc.driver.OracleDriver"
        url="jdbc:oracle:thin:@localhost:1521:xe"
        username="system" password="manager"
        maxTotal="20" maxIdle="10"
        maxWaitMillis="-1"/>
```

Step 2: Add the following entries application.properties to work with server managed JDBC connection pool.

application.properties

```
#To work with server managed JDBC connection pool
spring.datasource.jndi-name=java:/comp/env/DsJndi
```

Note: Server managed JDBC connection pool is possible only while working with external servers not with Spring Boot supplied embedded servers.

Form Validations in Spring Boot MVC Application

Best practice: Write both client side and server-side form validations but enable server-side form validations only when client-side form validations are not done.

- We generally write server-side form validations logics using Java code, and client-side form validations logics using JavaScript code.

Procedure to perform Server-side form validations

- Using JEE, Hibernate Validator Annotations
  - Enabling server form validations only when client-side form validations are not done is possible here.
  - Reusability of form validation logic across the multiple forms is not possible

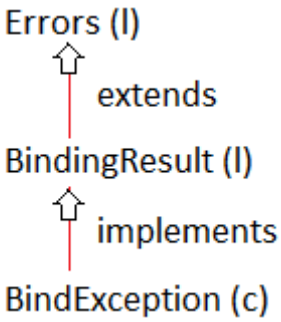
- Using separate Validator class
  - Validator class is the class that implement org.sf.validn.Validator (I) and we place validation logic inside the validate(-) method.

- Here enabling server form validation logics only when client form validations are not done is possible.
- Reusability of form validation logic across the multiple form pages is possible.

Server-side Form validations using Validator class

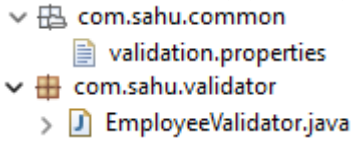
Step 1: Write form validation error messages in separate properties file and configure the custom properties file in application.properties file.
Step 2: Develop the validator class.
Step 3: Inject Validator class object to Controller class object using @Autowired.
Step 4: Write the code in insertEmployee(-,-), editEmployee(-,-) handler methods of Controller class for validation check.
Step 5: Write <form:errors path="*"/> inside <form:form> tags of add_employee.jsp, modify_employee.jsp form pages.

Note: Errors errors, and BindingResult errors refers to BindException object which maintain form validation error messages, Binding error messages and other messages.



Directory Structure of BootMVCProj10-MiniProject-FormValidation:

- Copy and paste the BootMVCProj09-MiniProject-SoftDeletion and rename to BootMVCProj10-MiniProject-FormValidation.
- After that change the Web Project Setting context root to BootMVCProj10-MiniProject-FormValidation.
- Then change in <artifactId> & <name> tag of pom.xml.
- Add the following package, class and file in the project.



- Then place the following code with in their respective files.

application.properties

```
#Configure user define property file
spring.messages.basename=com/sahu/common/validation
```

validation.properties

```
#From validation Error message (Server side) [insert, edit from)
emp.ename.required=Employee Name is required
emp.ename.minilength=Employee Name must have minimum 5 characters.
emp.desg.required=Employee Designation is required
emp.desg.minilength=Employee Designation must have minimum 5
characters.
emp.sal.required=Employee Salary is required
emp.sal.numeric=Employee Salary must be numeric value
emp.sal.range=Employee Salary must be in the range of 10000 to 100000
emp.detpno.required=Employee Dept.No is required
emp.deptno.numeric=Employee Dept.No must be numeric value
emp.deptno.range=Employee Dept.No must be in the range of 10 to 100000
```

EmployeeValidator.java

```
package com.sahu.validator;

import org.springframework.stereotype.Component;
import org.springframework.validation.Errors;
import org.springframework.validation.Validator;

import com.sahu.model.Employee;

@Component
public class EmployeeValidator implements Validator {

	/**
	 * This method is given to check weather handler method is passing
	 * correct command class object or not.
	 * If passed it calls validate(-,-) method other wise throws exception.
	 */
	@Override
	public boolean supports(Class<?> clazz) {
		System.out.println("EmployeeValidator.supports()");
		//return Employee.class==clazz;
```

```
        return clazz.isAssignableFrom(Employee.class);
    }

    @Override
    public void validate(Object target, Errors errors) {
        System.out.println("EmployeeValidator.validate()");
        //Type casting
        Employee emp = (Employee) target;
        //server side form  validation logic
        if (emp.getEname()==null||emp.getEname().isBlank())
            errors.rejectValue("ename", "emp.ename.required");
        else if (emp.getEname().length()<5)
            errors.rejectValue("ename", "emp.ename.minilength");

        if (emp.getJob()==null||emp.getJob().isBlank())
            errors.rejectValue("job", "emp.desg.required");
        else if (emp.getJob().length()<5)
            errors.rejectValue("ename", "emp.desg.minilength");

        if (emp.getSal()==null)
            errors.rejectValue("sal", "emp.sal.required");
        else if (emp.getSal().isNaN())
            errors.rejectValue("sal", "emp.sal.numeric");
        else if (emp.getSal()<10000 || emp.getSal()>100000)
            errors.rejectValue("sal", "emp.sal.range");

        if (emp.getDeptno()==null)
            errors.rejectValue("deptno", "emp.detpno.required");
        else if (((Float)emp.getDeptno().floatValue()).isNaN())
            errors.rejectValue("deptno", "emp.detpno.numeric");
        else if (emp.getDeptno()<10 || emp.getDeptno()>100)
            errors.rejectValue("deptno", "emp.detpno.range");
    }

}
```

add_employee.jsp

```
<form:form modelAttribute="emp">
    <p style="color: red; text-align: center;">
        <form:errors path="*"/>
    </p>
```

modify_employee.jsp

```
<form:form modelAttribute="emp">
	<p style="color: red; text-align: center;">
		<form:errors path="*"/>
	</p>
```

EmployeeController.java

```
@Controller
public class EmployeeController {

	@Autowired
	private IEmployeeMgmtService employeeMgmtService;

	@Autowired
	private EmployeeValidator employeeValidator;

	@PostMapping("/add_employee")
	public String insertEmployee(HttpSession session,
@ModelAttribute("emp") Employee emp, BindingResult bindingResult) {
		if (employeeValidator.supports(emp.getClass())) {
			employeeValidator.validate(emp, bindingResult);

			if (bindingResult.hasErrors())
				return "add_employee";
		}
		//Use service
		String result = employeeMgmtService.insertEmployee(emp);
		//Add result to HttpSession object
		session.setAttribute("resultMsg", result);
		//return LVN
		return "redirect:emp_report";
	}

	@PostMapping("/edit_employee")
	public String editEmployee(RedirectAttributes attrs,
@ModelAttribute("emp") Employee emp, BindingResult bindingResult) {
		if (employeeValidator.supports(emp.getClass())) {
			employeeValidator.validate(emp, bindingResult);
```

```
                    if (bindingResult.hasErrors())
                            return "modify_employee";
                }
                //Use service
                String result = employeeMgmtService.updateEmployee(emp);
                //Add result to RedirectAttributes object
                attrs.addFlashAttribute("resultMsg", result);
                //return LVN
                return "redirect:emp_report";
        }

}
```

To display form validation error messages beside the form components of form page we need to use <form:errors> tag by specifying field name as the "path" as shown below

add_employee.jsp

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

<h1 style="color: blue; text-align: center;">Register Employee</h1>

<form:form modelAttribute="emp">
        <%-- <p style="color: red; text-align: center;">
                <form:errors path="*"/>
        </p> --%>
        <table border="0" bgcolor="cyan" align="center">
                <tr>
                        <td>Employee Name : </td>
                        <td>
                                <form:input path="ename"/>
                                <form:errors path="ename" cssStyle="color:red"/>
                        </td>
                </tr>
                <tr>
                        <td>Employee Designation : </td>
                        <td>
```

```
                        <form:input path="job"/>
                        <form:errors path="job" cssStyle="color:red"/>
                    </td>
                </tr>
                <tr>
                    <td>Employee Salary : </td>
                    <td>
                        <form:input path="sal"/>
                        <form:errors path="sal" cssStyle="color:red"/>
                    </td>
                </tr>
                <tr>
                    <td>Employee Dept No : </td>
                    <td>
                        <form:input path="deptno"/>
                        <form:errors path="deptno" cssStyle="color:red"/>
                    </td>
                </tr>
                <tr>
                    <td></td>
                    <td>
                        <button type="reset">
                            <img class="operation-btn" alt="rest image"
src="images/reset.png">
                        </button>
                          
                        <input class="operation-btn" type="image"
src="images/submit.png">
                    </td>
                </tr>
        </table>
</form:form>

<style>
.operation-btn{
        width: 50px;
        height: 50px;
}
</style>
```

modify_employee.jsp

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

<h1 style="color: blue; text-align: center;">Edit Employee</h1>

<form:form modelAttribute="emp">
	<%-- <p style="color: red; text-align: center;">
		<form:errors path="*"/>
	</p> --%>
	<table border="0" bgcolor="cyan" align="center">
		<tr>
			<td>Employee No : </td>
			<td><form:input path="empno" readonly="true"/></td>
		</tr>
		<tr>
			<td>Employee Name : </td>
			<td>
				<form:input path="ename"/>
				<form:errors path="ename" cssStyle="color:red"/>
			</td>
		</tr>
		<tr>
			<td>Employee Designation : </td>
			<td>
				<form:input path="job"/>
				<form:errors path="job" cssStyle="color:red"/>
			</td>
		</tr>
		<tr>
			<td>Employee Salary : </td>
			<td>
				<form:input path="sal"/>
				<form:errors path="sal" cssStyle="color:red"/>
			</td>
		</tr>
		<tr>
			<td>Employee Dept No : </td>
			<td>
```

```
                        <form:input path="deptno"/>
                        <form:errors path="deptno" cssStyle="color:red"/>
                    </td>
                </tr>
                <tr>
                    <td></td>
                    <td>
                        <button type="reset">
                            <img class="operation-btn" alt="rest image"
src="images/cancel.png">
                        </button>
                          
                        <input class="operation-btn" type="image"
src="images/update-user.png">
                    </td>
                </tr>
        </table>
</form:form>

<style>
.operation-btn{
        width: 50px;
        height: 50px;
}
</style>
```

- In Errors object (BindException object) we can have 3 types of error messages

a. Form Validation error messages
b. Type Mismatch error messages
c. Application Logic error messages

Type Mismatch error messages

- While binding form components data to model class object attributes/ properties/ fields the necessary conversion takes automatically by using Built-in property editors.
- If conversion is not passible then binding related type mismatch error will be displayed as technical message which is not suggested to display.
- We can customize this message as type mismatch error message using

typeMismatch.<property-name>=<value> message of properties file. (fixed)

validation.properties

```
#TypeMismatch errors
typeMismatch.sal=Employee Salary must be numeric value
typeMismatch.deptno=Employee Dept.No must be numeric value
```

Application Logic error messages

- These error messages are related business logic execution not allowing BSNL series phone numbers, not allowed certain address, not allowing certain designation and etc. in registration or modification process the record.

Step 1: Add application logic error message in the properties file.

validation.properties

```
#Application logic Error message
emp.job.restriction=Employee designation must not be President or QUEEN.
```

Step 2: In the POST back handler methods of controller class, keep the logics to generate application logic errors. In insertEmployee(-,-,-) or editEmployee(-,-,-) methods of EmployeeController.java class.

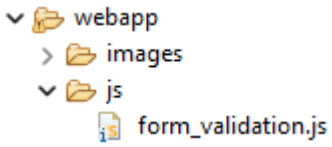
EmployeeController.java

```
		//Application/ business logic errors
		if (emp.getJob().equalsIgnoreCase("PRESIDENT") ||
emp.getJob().equalsIgnoreCase("QUEEN")) {
			bindingResult.rejectValue("job", "emp.job.restriction");
			return "modify_employee";
		}
```

- The order of storing error Messages in Errors object is
  a. Type Mismatch error messages
  b. Form Validation error messages
  c. Application Logic error messages

Client-Side form validations

Step 1: Develop separate .js file having form validation logic webapp\js\form_validation.js.



form_validation.js

```
function validate(frm) {
	//Read form data
	let name = frm.ename.value;
	let job = frm.job.value;
	let sal = frm.sal.value;
	let deptno = frm.deptno.value;
	let validationFlag =  true;

	//form validation logics (client side)
	if(name=="") {
		document.getElementById("enameErr").innerHTML="Employee name is required";
		validationFlag =false;
	}
	else if(name.length()<5){
		document.getElementById("enameErr").innerHTML="Employee name must have minimum of 5 character";
		validationFlag=false;
	}

	if(job=="") {
		document.getElementById("jobErr").innerHTML="Employee designation is required";
		validationFlag =false;
	}
	else if(job.length()<5){
		document.getElementById("jobErr").innerHTML="Employee designation must have minimum of 5 character";
		validationFlag =false;
	}
```

```
        if(sal=="") {
                document.getElementById("salErr").innerHTML="Employee
salary is required";
                validationFlag =false;
        }
        else if(isNaN(sal)){
                document.getElementById("salErr").innerHTML="Employee
salary must have be numberic value";
                validationFlag =false;
        }
        else if(sal<10000 || sal>100000){
                document.getElementById("salErr").innerHTML="Employee
salary must have be given in the range of 10000 to 100000";
                validationFlag =false;
        }

        if(deptno=="") {

        document.getElementById("deptnoErr").innerHTML="Employee Dept
no is required";
                validationFlag =false;
        }
        else if(isNaN(deptno)){

        document.getElementById("deptnoErr").innerHTML="Employee Dept
no must have be numberic value";
                validationFlag =false;
        }
        else if(deptno<10000 || deptno>100000){

        document.getElementById("deptnoErr").innerHTML="Employee Dept
no must have be given in the range of 10 to 100";
                validationFlag =false;
        }

        return validationFlag;
}
```

Step 2: Import and use JavaScript code in form pages (add_employee.jsp and edit_employee.jsp).

add_employee.jsp

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

<h1 style="color: blue; text-align: center;">Register Employee</h1>

<script type="text/javascript" src="js/form_validation.js"></script>

<form:form modelAttribute="emp" onsubmit="return validate(this)">
	<%-- <p style="color: red; text-align: center;">
		<form:errors path="*"/>
	</p> --%>
	<table border="0" bgcolor="cyan" align="center">
		<tr>
			<td>Employee Name : </td>
			<td>
				<form:input path="ename"/>
				<form:errors path="ename" cssStyle="color:red"/>
				<span id="enameErr" style="color: red;"></span>
			</td>
		</tr>
		<tr>
			<td>Employee Designation : </td>
			<td>
				<form:input path="job"/>
				<form:errors path="job" cssStyle="color:red"/>
				<span id="jobErr" style="color: red;"></span>
			</td>
		</tr>
		<tr>
			<td>Employee Salary : </td>
			<td>
				<form:input path="sal"/>
				<form:errors path="sal" cssStyle="color:red"/>
				<span id="salErr" style="color: red;"></span>
			</td>
		</tr>
		<tr>
			<td>Employee Dept No : </td>
```

```
                    <td>
                        <form:input path="deptno"/>
                        <form:errors path="deptno" cssStyle="color:red"/>
                        <span id="deptnoErr" style="color: red;"></span>
                    </td>
                </tr>
                <tr>
                    <td></td>
                    <td>
                        <button type="reset">
                            <img class="operation-btn" alt="rest image"
src="images/reset.png">
                        </button>
                          
                        <input class="operation-btn" type="image"
src="images/submit.png">
                    </td>
                </tr>
        </table>
</form:form>

<style>
.operation-btn{
        width: 50px;
        height: 50px;
}
</style>
```

modify_employee.jsp

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

<h1 style="color: blue; text-align: center;">Edit Employee</h1>

<script type="text/javascript" src="js/form_validation.js"></script>

<form:form modelAttribute="emp"  onsubmit="return validate(this)">
        <%-- <p style="color: red; text-align: center;">
                <form:errors path="*" />
```

```
        <form:errors path="*"/>
    </p> --%>
    <table border="0" bgcolor="cyan" align="center">
        <tr>
            <td>Employee No : </td>
            <td><form:input path="empno" readonly="true"/></td>
        </tr>
        <tr>
            <td>Employee Name : </td>
            <td>
                <form:input path="ename"/>
                <form:errors path="ename" cssStyle="color:red"/>
                <span id="enameErr"  style="color: red;"></span>
            </td>
        </tr>
        <tr>
            <td>Employee Designation : </td>
            <td>
                <form:input path="job"/>
                <form:errors path="job" cssStyle="color:red"/>
                <span id="jobErr" style="color: red;"></span>
            </td>
        </tr>
        <tr>
            <td>Employee Salary : </td>
            <td>
                <form:input path="sal"/>
                <form:errors path="sal" cssStyle="color:red"/>
                <span id="salErr" style="color: red;"></span>
            </td>
        </tr>
        <tr>
            <td>Employee Dept No : </td>
            <td>
                <form:input path="deptno"/>
                <form:errors path="deptno" cssStyle="color:red"/>
                <span id="deptnoErr" style="color: red;"></span>
            </td>
        </tr>
        <tr>
```

```
                    <td></td>
                    <td>
                        <button type="reset">
                            <img class="operation-btn" alt="rest image"
src="images/cancel.png">
                        </button>
                          
                        <input class="operation-btn" type="image"
src="images/update-user.png">
                    </td>
                </tr>
        </table>
</form:form>
<style>
.operation-btn{
        width: 50px;
        height: 50px;
}
</style>
```

Enabling Server-side form validations only when Client-side (JavaScript) form validations are not done

Step 1: Take additional @Trasient property in model class to carry flag value indicating whether client-side form validations are done or not.

Employee.java

```
        @Transient
        private String vflag="no";
```

Step 2: Take hidden box in the form page to hold vflag property default value

add_employee.jsp

```
<form:form modelAttribute="emp" onsubmit="return validate(this)">
        <form:hidden path="vflag"/>
```

modify_employee.jsp

```
<form:form modelAttribute="emp" onsubmit="return validate(this)">
        <form:hidden path="vflag"/>
```

Step 3: Write additional code in JavaScript to change the hidden box vflag value to "yes" indicating client-side form validation are performed.

form_validation.js

```
	// changing vflag (hidden box value 'yes' indicating client-side form validations are performed)
	frm.vflag.value="yes";

	return validationFlag;
```

Step 4: Write following code in insertEmployee(-,-,-) , editEmployee(-,-,-) methods to execute server side form validation logic only client-side form validations are not done.

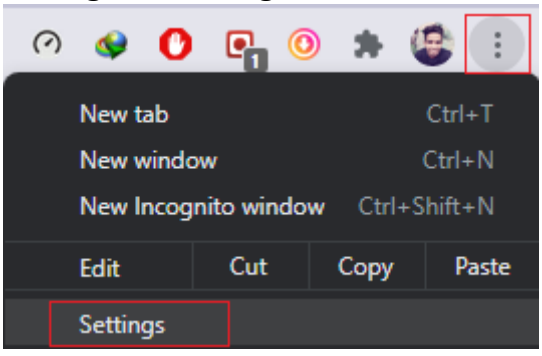
EmployeeController.java

```
	@PostMapping("/add_employee")
	public String insertEmployee(HttpSession session, @ModelAttribute("emp") Employee emp, BindingResult bindingResult) {
		if(emp.getVflag().equalsIgnoreCase("no")) {
			if (employeeValidator.supports(emp.getClass())) {
				employeeValidator.validate(emp, bindingResult);

				if (bindingResult.hasErrors())
					return "add_employee";
			}
		}

		//Application/ business logic errors
		if (emp.getJob().equalsIgnoreCase("PRESIDENT") || emp.getJob().equalsIgnoreCase("QUEEN")) {
			bindingResult.rejectValue("job", "emp.job.restriction");
			return "add_employee";
		}

		//Use service
		String result = employeeMgmtService.insertEmployee(emp);
		//Add result to HttpSession object
		session.setAttribute("resultMsg", result);
		//return LVN
```

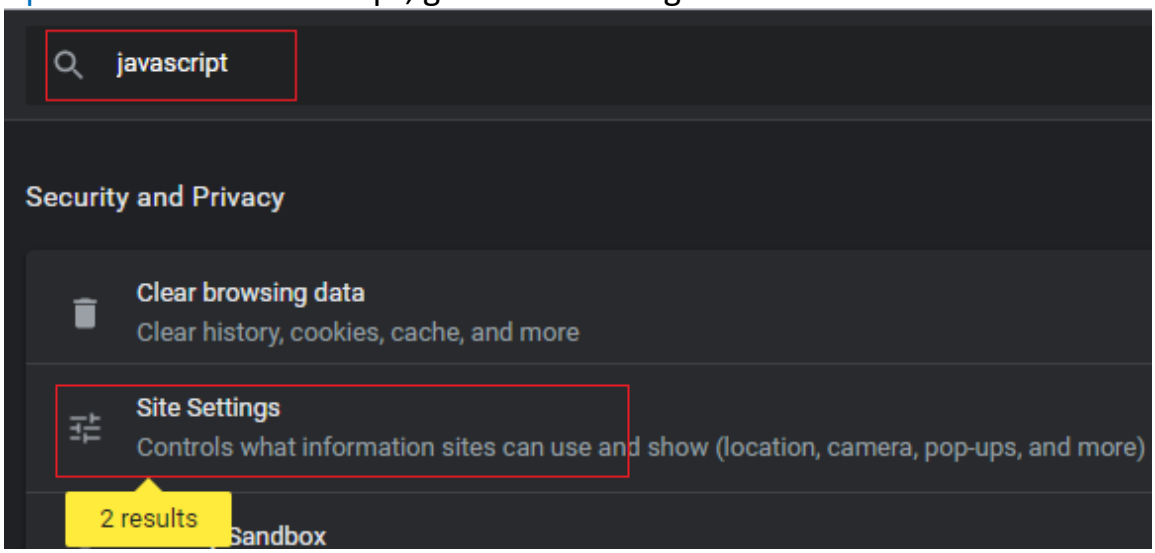
```
            return "redirect:emp_report";
        }

        @PostMapping("/edit_employee")
        public String editEmployee(RedirectAttributes attrs,
@ModelAttribute("emp") Employee emp, BindingResult bindingResult) {
            if(emp.getVflag().equalsIgnoreCase("no")) {
                if (employeeValidator.supports(emp.getClass())) {
                    employeeValidator.validate(emp, bindingResult);

                    if (bindingResult.hasErrors())
                        return "modify_employee";
                }
            }

            //Application/ business logic errors
            if (emp.getJob().equalsIgnoreCase("PRESIDENT") ||
emp.getJob().equalsIgnoreCase("QUEEN")) {
                bindingResult.rejectValue("job", "emp.job.restriction");
                return "modify_employee";
            }

            //Use service
            String result = employeeMgmtService.updateEmployee(emp);
            //Add result to RedirectAttributes object
            attrs.addFlashAttribute("resultMsg", result);
            //return LVN
            return "redirect:emp_report";
        }
```

Note: To enable or disable Javascript code using chrome browser follow the below steps.

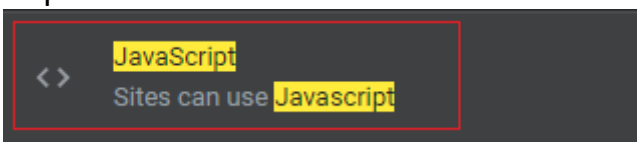
Step 1: Click on menu then go to settings.



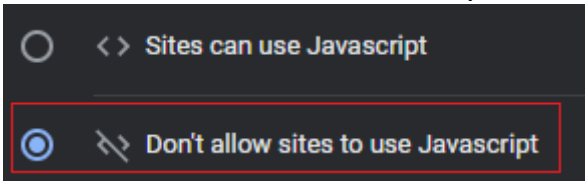
Step 2: Search for JavaScript, go to Site Settings.



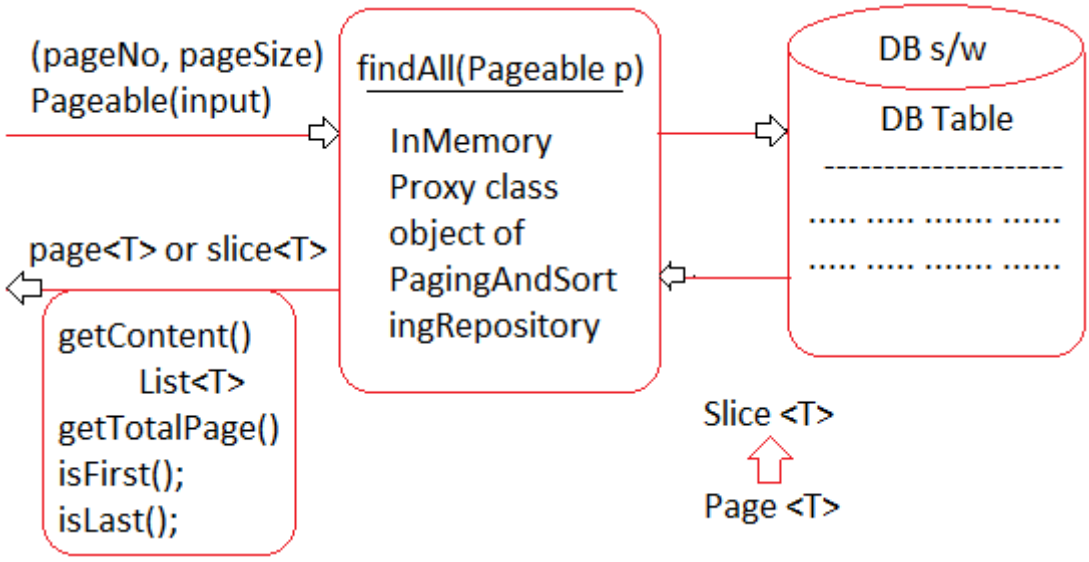
Step 3: Go to JavaScript.



Step 4: Choose the Don't allow sites to use JavaScript.



Spring Boot MVC with Spring Data JPA Pagination



- Page<T> findAll (Pageable pageable); this method takes pageNo (0

based), pageSize as inputs in the form of Pageable object and returns output as Page<T>/ Slice<T> object having requested page records, pages count, current numbers, total records and etc.

- If total records count is 20 and pagesize is 5 then it divides 20 records into 4 pages (5, 5, 5, 5). If we ask for 3rd page (indirectly 4th page) records then it gives 16 to 20 records. If ask for 2nd page (indirectly 3rd page) records then it gives 11 to 15 records.
- Pageable pageable = PageRequest.of(2,5);

Note: Pageable object can have both Paging and Sorting information.

- DispatcherServlet can prepare Pageable object and can give as handler method argument value if the parameter type is Pageable. In this process it prepares that Pageable object having pageNo, pageSize. gathered from the request parameters "page", "size".
- We can give default pageNo, pageSize to the Pageable object of Handler method parameter using @PageableDefault annotation as shown below.

```
@GetMapping("/emp_report")
public String showEmployeeReport( @PageableDefault(page=0, size=3,
            sort="job", direction=Sort.Direction.ASC) Pageable pageable,
                                                Map<String,Object> map) {
        ……….
        …….
}
```

IEmployeeMgmtService.java

```
public Page<Employee> getEmployeesPageData(Pageable pageable);
```

EmployeeMgmtServiceImpl.java

```
@Override
public Page<Employee> getEmployeesPageData(Pageable pageable) {
        return employeeRepo.findAll(pageable);
}
```

EmployeeMgmtServiceImpl.java

```
@GetMapping("/emp_report")
public String showEmployeeReport(@PageableDefault(page = 0, size
= 3, sort = "job", direction = Direction.ASC) Pageable pageable, Map<String,
```

```
Object> map) {
                //User Service
                Page<Employee> page =
employeeMgmtService.getEmployeesPageData(pageable);
                //Keep results as model attribute
                map.put("pageData", page);
                //return LVN
                return "employee_report";
        }
```

employee_report.jsp

```
<%@ page isELIgnored="false"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>

<c:choose>
        <c:when test="${!empty pageData}">
                <table border="1" bgcolor="cyan" align="center">
                        <tr bgcolor="pink">
                                <th>ENo</th>
                                <th>EName</th>
                                <th>Desg</th>
                                <th>Salary</th>
                                <th>Dept No</th>
                                <th>Operations</th>
                        </tr>
                        <c:forEach var="emp"
items="${pageData.getContent()}">
                                <tr>
                                        <td>${emp.empno}</td>
                                        <td>${emp.ename}</td>
                                        <td>${emp.job}</td>
                                        <td>${emp.sal}</td>
                                        <td>${emp.deptno}</td>
                                        <td>
                                                <a
href="edit_employee?eno=${emp.empno}">
                                                        <img class="user-operation"
src="images/edit-user.png" />
                                                </a>   
                                                <a
```

```
href="delete_employee?eno=${emp.empno}" onclick="confirm('Do you
want to delete?')">
                                        <img class="user-operation"
src="images/delete-user.png" />
                                    </a>
                                </td>
                            </tr>
                        </c:forEach>
                    </table>
                    <br><br>
                    <div style="text-align: center;">
                        <c:if test="${!pageData.isFirst()}">
                            <a
href="emp_report?page=0">[First]</a>  
                        </c:if>

                        <c:if test="${!pageData.isLast()}">
                            <a
href="emp_report?page=${pageData.getNumber()+1}">[Next]</a> &
nbsp;
                        </c:if>

                        <c:forEach var="i" begin="1"
end="${pageData.getTotalPages()}" step="1">
                            [<a href="emp_report?page=${i-
1}">${i}</a>]  
                        </c:forEach>

                        <c:if test="${!pageData.isFirst()}">
                            <a
href="emp_report?page=${pageData.getNumber()-
1}">[Previous]</a>  
                        </c:if>

                        <c:if test="${!pageData.isLast()}">
                            <a
href="emp_report?page=${pageData.getTotalPages()-1}">[Last]</a>
                        </c:if>
                    </div>
                </c:when>
```

```
        <c:otherwise>
                <h1 style="color: red; text-align: center;">Records not
Found</h1>
        </c:otherwise>
</c:choose>
<br>
        <h1 class="blink_me" style="color: green; text-align:
center;">${resultMsg}</h1>
<br>
<div style="text-align: center;">
        <a href="add_employee">
                <img class="user-operation" src="images/add-user.png" />
        </a>
          
        <a href="./">
                <img class="user-operation" src="images/home.png" />
        </a>
</div>

<style>
.user-operation {
        height: 50px;
        width: 50px;
}

.blink_me {
  animation: blinker 1s linear infinite;
}

@keyframes blinker {
  50% {
    opacity: 0;
  }
}
</style>
```

[Next] [1] [2] [3] [4] [5] [Last]

Reference Data

- While working with special form components like radio buttons group, check boxes group, select box, list box we need this Reference data concept using @ModelAttribute annotation.

Form Page	In Model class
Gender: ○ male ● female gender	private String gender = “female”;
Hobbies: ☐ reading ☐ writing ☑ watching TV ☑ playing ☐ sleeping ☑ travelling	private String [] hobbies;
Hobbies: ☐ reading ☐ writing ☑ watching TV ☑ playing ☐ sleeping ☑ travelling	private String country = “india”;

- The above 4 special components (grouped radio buttons, group check boxes, select box, list box elements/ items cannot be collected and displayed from their respective model class object properties because their respective model class properties are designed to hold selected items, not all the items which are required to display in the form. (Especially if you are displaying them dynamically by collecting through service logics and from DB s/w).
- We need to construct these items @ModelAttribute methods in Controller and such data is called Reference data.

@ModelAttribute is multipurpose Annotation:

a. Useful to make Java bean class as Model class supporting two-way binding of form data.
b. Useful to construct dynamic reference data that is required as the items/ elements for the four special components like grouped check boxes, grouped radio buttons, select box and list box components.

Q. What is the different b/w <form:option> and <form:options> tags?

Ans.

- <form:option> tag is given to place hardcoded items in select box, list box components.
- <form:options> tag is given to place dynamic items collected from the specified model attribute in the select box, list box components.

- E.g.,

```
<form:select path="country">
        <form:option value="india">India</form:option>
        <form:option value="china">China </form:option>
        ……
</form:select>
                (or)
<form:select path="country">
        <form:options items="${countriesInfo}"/>
</form:select>
```

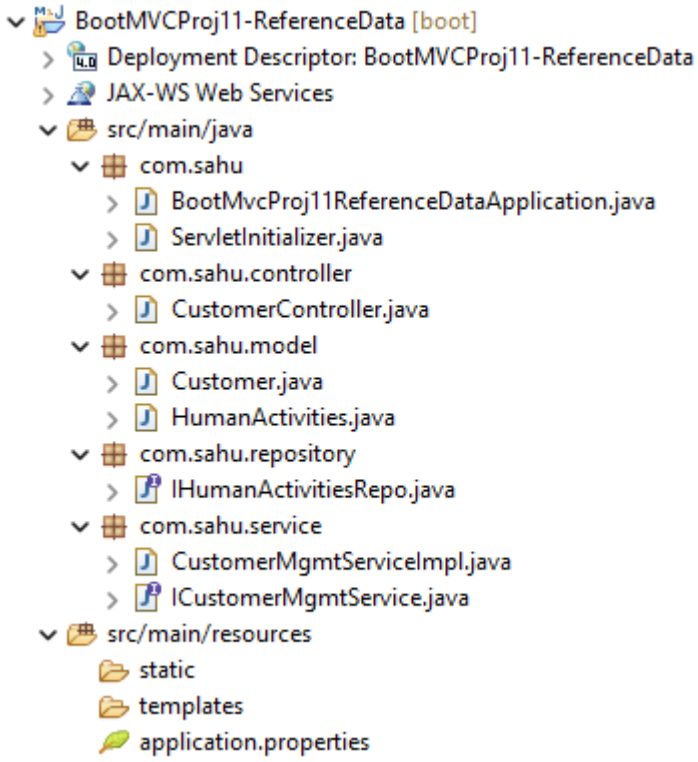
Q. What is the difference <form:checkbox> and <form:checkboxes> tags?

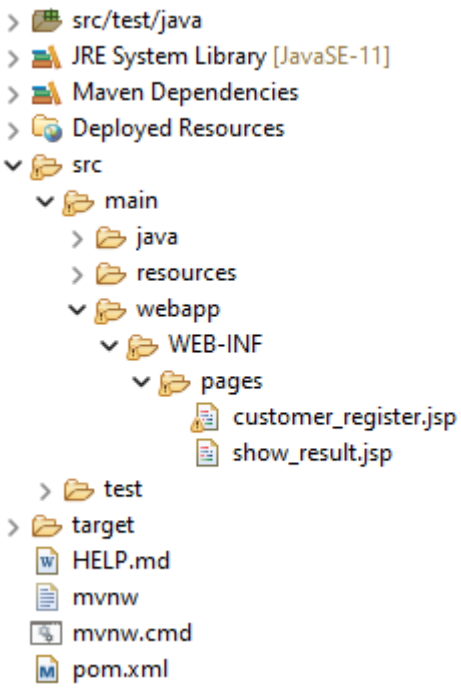
Ans.

- <form:checkbox> gives checkbox item with hardcoded value and label.
- <form:checkboxes> give dynamic checkbox items gathered from reference data given by Model Attributes.
- E.g.,

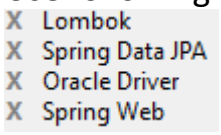
```
<form:checkboxe value="watchingtv" path="hobbies"/> Watching TV
<form:checkboxe value="reading" path="hobbies"/> Reading Books
                        (or)
<form:checkboxes items="${hobbiesInfo}" path="hobbies"/>
```

Directory Structure of BootMVCProj11-ReferenceData:





- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also
- Use following starters during project creation.



- Collect application.properties file from previous project and remove unnecessary properties.
- Then place the following code with in their respective files.

Customer.java

```
package com.sahu.model;

import lombok.Data;

@Data
public class Customer {
	private Integer cno;
	private String cname;
	private String country="India";
	private String[] languages=new String[] {"English", "Hindi"};
	private String hobbies[] = new String[] {"Swimming", "Workout"};
}
```

HumanActivities.java

```
package com.sahu.model;

import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;

import lombok.Data;

@Entity
@Data
@Table(name = "HUMAN_ACTIVITIES")
public class HumanActivities {
	@Id
	private String hobbies;
}
```

IHumanActivitiesRepo.java

```
package com.sahu.repository;

import java.util.Set;

import org.springframework.data.jpa.repository.Query;
import org.springframework.data.repository.CrudRepository;

import com.sahu.model.HumanActivities;

public interface IHumanActivitiesRepo extends
CrudRepository<HumanActivities, String> {
	@Query("SELECT hobbies FROM HumanActivities")
	public Set<String> getAllHobbies();
}
```

ICustomerMgmtService.java

```
package com.sahu.service;

import java.util.Set;

public interface ICustomerMgmtService {
```

```
        public Set<String> getAllCountries();
        public Set<String> getAllLanguages();
        public Set<String> getAllHobbies();
}
```

IHumanActivitiesRepo.java

```
package com.sahu.service;

import java.util.Locale;
import java.util.Set;
import java.util.TreeSet;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.repository.IHumanActivitiesRepo;

@Service("custService")
public class CustomerMgmtServiceImpl implements ICustomerMgmtService
{

        @Autowired
        private IHumanActivitiesRepo humanActivitiesRepo;;

        @Override
        public Set<String> getAllCountries() {
                Locale[] locales = Locale.getAvailableLocales();
                Set<String> countriesSet = new TreeSet<>();
                for (Locale locale : locales)
                        countriesSet.add(locale.getDisplayCountry());

                return countriesSet;
        }

        @Override
        public Set<String> getAllLanguages() {
                Locale[] locales = Locale.getAvailableLocales();
                Set<String> languagesSet = new TreeSet<>();
                for (Locale locale : locales)
                        languagesSet.add(locale.getDisplayLanguage());
```

```
		return languagesSet;
	}

	@Override
	public Set<String> getAllHobbies() {
		return humanActivitiesRepo.getAllHobbies();
	}

}
```

CustomerController.java

```
package com.sahu.controller;

import java.util.Map;
import java.util.Set;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

import com.sahu.model.Customer;
import com.sahu.service.ICustomerMgmtService;

@Controller
public class CustomerController {

	@Autowired
	private ICustomerMgmtService customerMgmtService;

	@GetMapping("/")
	public String showCustomerFormPage(@ModelAttribute("cust")
Customer cust) {
		return "customer_register";
	}

	@PostMapping("/register_customer")
	public String registerCustomer(Map<String, Object> map,
```

```
@ModelAttribute("cust") Customer cust) {
		return "show_result";
	}

	@ModelAttribute("countriesInfo")
	public Set<String> populateCountries(){
		return customerMgmtService.getAllCountries();
	}

	@ModelAttribute("languagesInfo")
	public Set<String> populateLanguages(){
		return customerMgmtService.getAllLanguages();
	}

	@ModelAttribute("hobbiesInfo")
	public Set<String> populateHobbies(){
		return customerMgmtService.getAllHobbies();
	}

}
```

customer_register.jsp

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

<form:form action="register_customer" modelAttribute="cust">
	<table align="center" bgcolor="cyan">
		<tr>
			<td>Customer Name: </td>
			<td><form:input path="cname"/></td>
		</tr>
		<tr>
			<td>Select Country: </td>
			<td>
				<form:select path="country">
					<form:options items="${countriesInfo}"/>
				</form:select>
			</td>
		</tr>
```

```
            <tr>
                <td>Select Known Languages: </td>
                <td>
                    <form:select path="languages"
multiple="multiple" size="5">
                        <form:options items="${languagesInfo}"/>
                    </form:select>
                </td>
            </tr>
            <tr>
                <td>Select Hobbies: </td>
                <td>
                    <form:checkboxes items="${hobbiesInfo}"
path="hobbies"/>
                </td>
            </tr>
            <tr>
                <td></td>
                <td>
                    <input type="reset" value="Cancel">  
                    <input type="submit" value="Register">
                </td>
            </tr>
        </table>
</form:form>
```

show_result.jsp

```
<%@ page isELIgnored="false"%>

<b>Model class Object Data/ Form data: ${cust}</b>
<br>
<a href="./">home</a>
```

Populating Dynamic content to second select box based on the item selected from the first select box

E.g., Getting states in to second select box based on the country selected from the first select box.

Step 1: Keep countries and states info in the application.properties file.

application.properties

```
India=Andhra Pradesh,Arunachal
Pradesh,Assam,Bihar,Chhattisgarh,Goa,Gujarat,Haryana,Himachal
Pradesh,Jharkhand,Karnataka,Kerala,Madhya
Pradesh,Maharashtra,Manipur,Meghalaya,Mizoram,Nagaland,Odisha,Punja
b,Rajasthan,Sikkim,Tamil Nadu,Telangana,Tripura,Uttarakhand,Uttar
Pradesh,West Bengal
United\u0020States=Alabama,Alaska,Arizona,Arkansas,California,Colorado,
Connecticut,Delaware,Florida,Georgia,Hawaii,Idaho,Illinois,Indiana,Iowa,Ka
nsas,Kentucky,Louisiana,Maine,Maryland,Massachusetts,Michigan,Minneso
ta,Mississippi,Missouri,Montana,Nebraska,Nevada,New Hampshire,New
Jersey,New Mexico,New York,North Carolina,North
Dakota,Ohio,Oklahoma,Oregon,Pennsylvania,Rhode Island,South
Carolina,South
Dakota,Tennessee,Texas,Utah,Vermont,Virginia,Washington,West
Virginia,Wisconsin,Wyoming
```

Step 2: Inject properties file info service implementation class as Environment object and use the content in the business method that gives list of states from the country that is selected.

ICustomerMgmtService.java

```
        public List<String> getStatesByCountry(String country);
```

ICustomerMgmtService.java

```
        @Autowired
        private Environment environment;

        @Override
        public List<String> getStatesByCountry(String country) {
                //String States from properties file using Environment object
                String statesInfo = environment.getRequiredProperty(country);
                List<String> statesList = Arrays.asList(statesInfo.split(","));
                return statesList;
        }
```

Step 3: Take property in model class for “state” select box component.

Customer.java

```
        private String state;
```

Step 4: Add the following code in form page to send request to on change event of country select box.

customer_register.jsp

```
<form:form name="frm" action="register_customer"
modelAttribute="cust">
        <table align="center" bgcolor="cyan">
                <tr>
                        <td>Customer Name: </td>
                        <td><form:input path="cname"/></td>
                </tr>
                <script type="text/javascript">
                        function sendReqForStates() {
                                frm.action="states";
                                frm.submit();
                        }
                </script>
                <tr>
                        <td>Select Country: </td>
                        <td>
                                <form:select path="country"
onchange="sendReqForStates()">
                                        <form:options items="${countriesInfo}"/>
                                </form:select>
                        </td>
                </tr>
                <tr>
                        <td>Select State: </td>
                        <td>
                                <form:select path="state">
                                        <form:options items="${statesInfo}"/>
                                </form:select>
                        </td>
                </tr>
```

Step 5: Add handler method in controller class.

CustomerController.java

```
        @PostMapping("/states")
        public String showStatesByCountry(@RequestParam("country")
String country, @ModelAttribute("cust") Customer cust, Map<String,
Object> map) {
                //Use service
                List<String> statesList
=customerMgmtService.getStatesByCountry(country);
                //add to model attribute
                map.put("statesInfo", statesList);
                //return LVN
                return "customer_register";
        }
```

Init Binder in Spring MVC

- In Spring we use the support Property Editors to convert given String values to different types of required value before performing injection or binding.
- All Property Editors are classes implementing PropertyEditor (I) directly or indirectly,
- Spring gives multiple Built-in Property Editors,

ByteArrayPropertyEditor	Editor for byte arrays.
CharacterEditor	Editor for a **Character**, to populate a property of type Character or char from a String value.
CharArrayPropertyEditor	Editor for char arrays.
CharsetEditor	Editor for java.nio.charset.Charset, translating charset String representations into Charset objects and back.
ClassArrayEditor	Property editor for an array of **Classes**, to enable the direct population of a Class[] property without having to use a String class name property as bridge.

ClassEditor	Property editor for **java.lang.Class**, to enable the direct population of a Class property without recourse to having to use a String class name property as bridge.
CurrencyEditor	Editor for java.util.Currency, translating currency codes into Currency objects.
CustomBooleanEditor	Property editor for Boolean/boolean properties.
CustomCollectionEditor	Property editor for Collections, converting any source Collection to a given target Collection type.
CustomDateEditor	Property editor for java.util.Date, supporting a custom java.text.DateFormat.
CustomMapEditor	Property editor for Maps, converting any source Map to a given target Map type.
CustomNumberEditor	Property editor for any Number subclass such as Short, Integer, Long, BigInteger, Float, Double, BigDecimal.
FileEditor	Editor for java.io.File, to directly populate a File property from a Spring resource location.
InputSourceEditor	Editor for org.xml.sax.InputSource, converting from a Spring resource location String to a SAX InputSource object.
InputStreamEditor	One-way PropertyEditor which can convert from a text String to a java.io.InputStream, interpreting the given String as a Spring resource location (e.g.
LocaleEditor	Editor for java.util.Locale, to directly populate a Locale property.

PathEditor	Editor for java.nio.file.Path, to directly populate a Path property instead of using a String property as bridge.
PatternEditor	Editor for java.util.regex.Pattern, to directly populate a Pattern property.
PropertiesEditor	Custom **PropertyEditor** for **Properties** objects.
ReaderEditor	One-way PropertyEditor which can convert from a text String to a java.io.Reader, interpreting the given String as a Spring resource location (e.g.
ResourceBundleEditor	**PropertyEditor** implementation for standard JDK **ResourceBundles**.
StringArrayPropertyEditor	Custom **PropertyEditor** for String arrays.
StringTrimmerEditor	Property editor that trims Strings.
TimeZoneEditor	Editor for java.util.TimeZone, translating timezone IDs into TimeZone objects.
URIEditor	Editor for java.net.URI, to directly populate a URI property instead of using a String property as bridge.
URLEditor	Editor for java.net.URL, to directly populate a URL property instead of using a String property as bridge.
UUIDEditor	Editor for java.util.UUID, translating UUID String representations into UUID objects and back.
ZoneIdEditor	Editor for java.time.ZoneId, translating zone ID Strings into ZoneId objects.

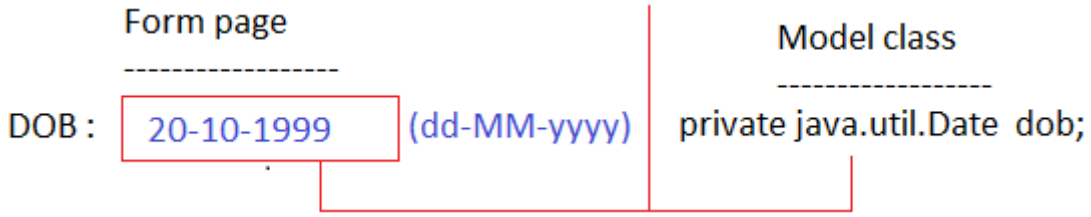
- Most of these property editors automatically registered with IOC

container and will be used internally towards data injection or data binding process.

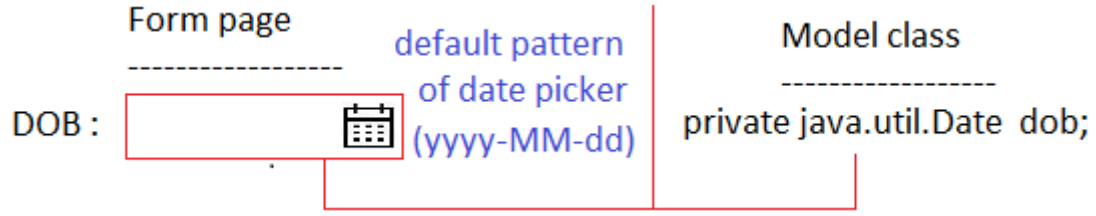
```
java.lang.Object
    org.springframework.validation.DataBinder
        org.springframework.web.bind.WebDataBinder
            org.springframework.web.bind.ServletRequestDataBinder
```

- DispatcherServlet internally uses ServletRequestDataBinder component to perform data binding activity i.e., writing form data to Model class object properties.
- This binder component internally uses multiple registered Property editors to convert the form components supplied string input values to different types values as required for the Model class object properties.

- ServletRequestDataBinder component internally uses one or another built-in property editor to convert (MM/dd/yyyy) patter String date value to java.util.Date class object.



- ServletRequestDataBinder component fails to convert dd-MM-yyyy pattern String date value to java.util.Date class object, so exception will be thrown. To overcome this problem, we need to register Property Editor with ServletRequestDataBinder component who can convert String date value (dd-MM-yyyy) to java.util.Date class object.



- ServletRequestDataBinder component fails to convert yyyy-MM-dd pattern String date value to java.util.Date class object, so exception will be thrown. To overcome this problem, we need to register Property Editor with ServletRequestDataBinder component who can convert String date value (yyyy-MM-dd) to java.util.Date class object.

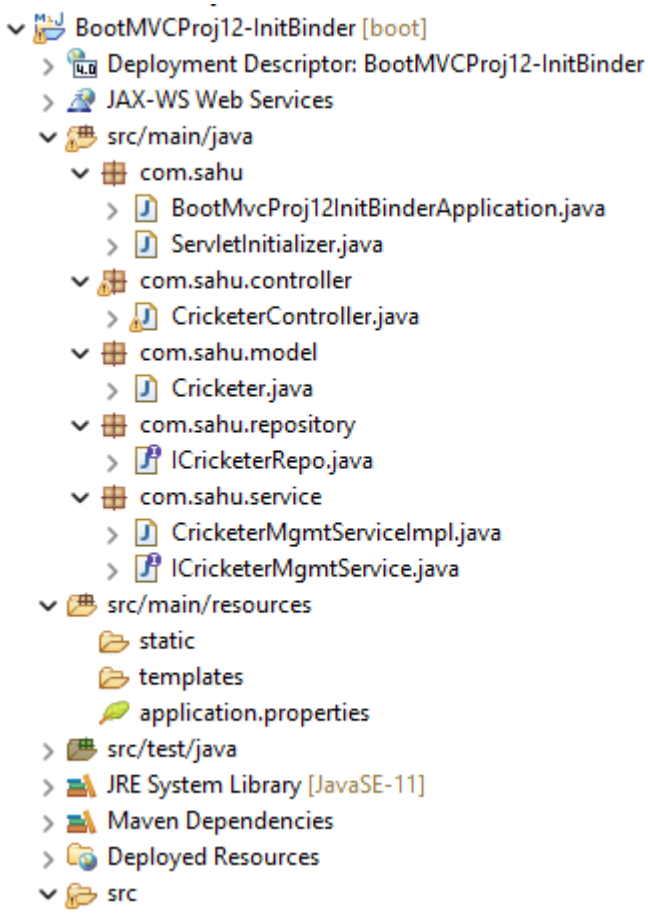
Note:

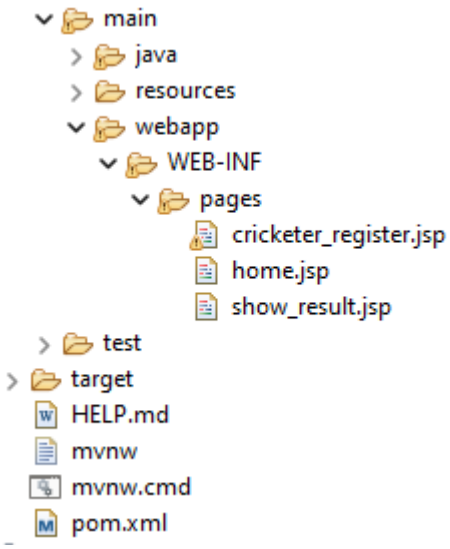
- ✓ To register Property Editors (readymade or custom) with ServletRequestDataBinder component we need to use @InitBinder methods.

```
@InitBinder
public void myBinder(WebDataBinder binder){
        binder.registerCustomEditor(-,-,-);
}
```

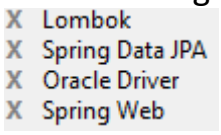
- ✓ @ModelAttrite method (reference data method) executes every time form launch.
- ✓ @InitBinder method executes for each form launching and form submission automatically.

Directory Structure of BootMVCProj12-InitBinder:





- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also
- Use following starters during project creation.



- Collect application.properties file from previous project and remove unnecessary properties.
- Then place the following code with in their respective files.

Cricketer.java

```
package com.sahu.model;

import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

import lombok.Data;

@Data
@Entity
@Table(name = "CRICKETER_INFO")
```

```
public class Cricketer {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Integer cid;
	@Column(length = 20)
	private String name;
	@Column(length = 20)
	private String type;
	private Date dob;
	private Date doj;
}
```

ICricketerRepo.java

```
package com.sahu.repository;

import org.springframework.data.repository.CrudRepository;

import com.sahu.model.Cricketer;

public interface ICricketerRepo extends CrudRepository<Cricketer, Integer>
{

}
```

ICricketerMgmtService.java

```
package com.sahu.service;

import com.sahu.model.Cricketer;

public interface ICricketerMgmtService {
	public String registerCricketer(Cricketer cricketer);
}
```

CricketerMgmtServiceImpl.java

```
package com.sahu.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
```

```
import com.sahu.model.Cricketer;
import com.sahu.repository.ICricketerRepo;

@Service
public class CricketerMgmtServiceImpl implements ICricketerMgmtService {

	@Autowired
	private ICricketerRepo cricketerRepo;

	@Override
	public String registerCricketer(Cricketer cricketer) {
		return cricketerRepo.save(cricketer).getName()+" detils has
registered";
	}

}
```

CricketerMgmtServiceImpl.java

```
package com.sahu.controller;

import java.text.SimpleDateFormat;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;

import com.sahu.model.Cricketer;
import com.sahu.service.ICricketerMgmtService;

@Controller
public class CricketerController {
```

```
	@Autowired
	private ICricketerMgmtService cricketerMgmtService;

	@GetMapping("/")
	public String showHomePage() {
		return "home";
	}

	@GetMapping("/register")
	public String
showCricketerRegistrationPage(@ModelAttribute("cktr")Cricketer cricketer)
{
		return "cricketer_register";
	}

	@PostMapping("/register")
	public String registerCricketer(Map<String, Object> map,
@ModelAttribute("cktr")Cricketer cktr, BindingResult errors) {
		if (errors.hasErrors())
			return "cricketer_register";
		//User service
		String message = cricketerMgmtService.registerCricketer(cktr);
		//Keep result in model attribute
		map.put("result", message);
		//return LVN
		return "show_result";
	}

	@InitBinder
	public void myInitBinder(WebDataBinder binder) {
		System.out.println("CricketerController.myInitBinder()");
		SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-
dd");
		//Ready Made PropertyEditor to convert String date value to
java.util.Date value
		CustomDateEditor editor = new CustomDateEditor(sdf, false);
		binder.registerCustomEditor(java.util.Date.class, editor);
	}

}
```

home.jsp

```
<%@ page isELIgnored="false" %>

<h1 style="color: red; text-align: center;">
	<a href="register">Register Cricketer</a>
</h1>
```

cricketer_register.jsp

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

<form:form modelAttribute="cktr">
	<table border="0" bgcolor="cyan" align="center">
		<p style="color: red;"><form:errors path="*"/></p>
		<tr>
			<td>Cricketer Name</td>
			<td><form:input path="name"/> </td>
		</tr>
		<tr>
			<td>Cricketer Type</td>
			<td><form:input path="type"/> </td>
		</tr>
		<tr>
			<td>Cricketer DOB</td>
			<td><form:input path="dob" type="date"/> </td>
		</tr>
		<tr>
			<td>Cricketer DOJ</td>
			<td><form:input path="doj" type="date"/> </td>
		</tr>
		<tr>
			<td></td>
			<td>
				<input type="reset" value="Cancel">  
				<input type="submit" value="Register">
			</td>
		</tr>
	</table>
</form:form>
```

show_result.jsp

```
<%@ page isELIgnored="false" %>

<h1 style="color: green; text-align: center;">
Register Message is: ${result}
</h1>
<br>
<a href="./">Home</a>
```

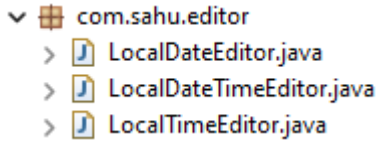
- CustomerDateEditor is pre-defined PropertyEditor class that is capable converting String date value of given pattern to java.util.Date class object, but it can't convert the same String date value to java.time.LocalDate or java.time.LocalDateTime class object.
- To overcome this problem we need to develop CustomPropertyEditor from scratch level either by implementing java.beans.PropertyEditor (I) or by extending from java.beans.PropertyEditorSupport (c).

The use cases we need to the support custom PropertyEditor:

- To convert String value to java.util.Locale object
- To convert String value to java.time.LocalTime object
- To convert String value to java.time.LocalDate object
- To convert String value to java.time.LocalDateTime object
- To convert String value to java.net.URL class object
  and etc.

Directory Structure of BootMVCProj13-InitBinder-LocalDateTime:

- Copy and paste the BootMVCProj12-InitBinder and rename to BootMVCProj13-InitBinder-LocalDateTime.
- After that change the Web Project Setting context root to BootMVCProj13-InitBinder-LocalDateTime.
- Then change in <artifactId> & <name> tag of pom.xml.
- Add the following package, class and file in the project.



- Then place the following code with in their respective files.

Note: Time datatype in not there in Oracle so you can try with MySQL for time.

Crickter.java

```
@Data
@Entity
@Table(name = "CRICKETER_INFO_LDT")
public class Cricketer {
	@Id
	@GeneratedValue(strategy = GenerationType.AUTO)
	private Integer cid;
	@Column(length = 20)
	private String name;
	@Column(length = 20)
	private String type;
	private LocalDate dob;
	private LocalTime tob;
	private LocalDateTime doj;
}
```

cricketer_register.jsp

```
		<tr>
			<td>Cricketer DOB</td>
			<td><form:input path="dob" type="date"/> </td>
		</tr>
		<tr>
			<td>Cricketer Time of Birth</td>
			<td><form:input path="tob" type="time"/> </td>
		</tr>
		<tr>
			<td>Cricketer Debue Date & Time</td>
			<td><form:input path="doj" type="datetime-local"/>
</td>
		</tr>
		<tr>
			<td></td>
			<td>
				<input type="reset" value="Cancel">  
				<input type="submit" value="Register">
			</td>
		</tr>
```

LocalDateEditor.java

```
package com.sahu.editor;

import java.beans.PropertyEditorSupport;
import java.time.LocalDate;

public class LocalDateEditor extends PropertyEditorSupport {

	@Override
	public void setAsText(String text) throws IllegalArgumentException {
		System.out.println("LocalDateEditor.setAsText()");
		//Split the String date value whose pattern is yyyy-MM-dd
		String dateContent[] = text.split("-");
		int year = Integer.parseInt(dateContent[0]);
		int month = Integer.parseInt(dateContent[1]);
		int day = Integer.parseInt(dateContent[2]);
		//Create LocalDate class object
		LocalDate localDate = LocalDate.of(year, month, day);
		//set value to model class property
		setValue(localDate);
	}

}
```

LocalDateTimeEditor.java

```
package com.sahu.editor;

import java.beans.PropertyEditorSupport;
import java.time.LocalDateTime;

public class LocalDateTimeEditor extends PropertyEditorSupport {

	@Override
	public void setAsText(String text) throws IllegalArgumentException {
		System.out.println("LocalDateTimeEditor.setAsText()");
		//Split the String date time value whose pattern is yyyy-MM-
ddThh:mm
		String content[] = text.split("T");
		String dateContent[] = content[0].split("-");
		String timeContent[] = content[1].split(":");
```

```
            int year = Integer.parseInt(dateContent[0]);
            int month = Integer.parseInt(dateContent[1]);
            int day = Integer.parseInt(dateContent[2]);

            int hours = Integer.parseInt(timeContent[0]);
            int minutes = Integer.parseInt(timeContent[1]);
            //Create LocalTime class object
            LocalDateTime localDateTime = LocalDateTime.of(year, month,
day, hours, minutes);
            //set value to model class property
            setValue(localDateTime);
        }

}
```

LocalTimeEditor.java

```
package com.sahu.editor;

import java.beans.PropertyEditorSupport;
import java.time.LocalTime;

public class LocalTimeEditor extends PropertyEditorSupport {

    @Override
    public void setAsText(String text) throws IllegalArgumentException {
        System.out.println("LocalTimeEditor.setAsText()");
        //Split the String time value whose pattern is hh:mm
        String timeContent[] = text.split(":");
        int hours = Integer.parseInt(timeContent[0]);
        int minutes = Integer.parseInt(timeContent[1]);
        //Create LocalTime class object
        LocalTime localTime = LocalTime.of(hours, minutes);
        //set value to model class property
        setValue(localTime);
    }

}
```

CricketerController.java

```
        @InitBinder
        public void myInitBinder(WebDataBinder binder) {
                System.out.println("CricketerController.myInitBinder()");
                binder.registerCustomEditor(LocalDate.class, new
LocalDateEditor());
                binder.registerCustomEditor(LocalTime.class, new
LocalTimeEditor());
                binder.registerCustomEditor(LocalDateTime.class, new
LocalDateTimeEditor());
        }
```

To run the same application in MySQL, follow the below steps:

- Add the MySQL starter in project.
- Then modify the following properties in application.properties file.

application.properties

```
#DataSource Configuration
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql:///ntspbms714db
spring.datasource.username=root
spring.datasource.password=root

#JPA Hibernate properties
spring.jpa.database-platform=org.hibernate.dialect.MySQL8Dialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=create
```

Handler Interceptors

- Handler interceptor acts as extension hook for controller class handler methods which allows us to put additional logics as pre, post, after completion logics to execute with respect to handler methods of controller classes execution.
- These are like Filters for Servlet/ JSP components of web application but filter allows only pre, post logics to place. Whereas interceptors are allowed us to place pre, post, after completion logic.
- To develop handler interceptor class, we need to make spring bean class implementing org.springframework.web.servlet.HandlerInterceptor (I).

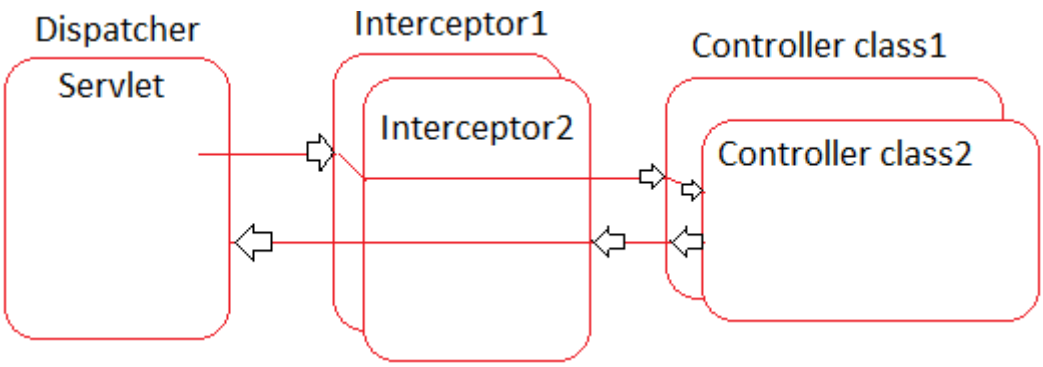
- Which has 3 default methods,
  1. **default boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) **throws** Exception
     - Executes before getting into handler method of controller class.
     - Use case: allowing requests in certain timings, allowing requests only from certain browsers and etc.

Note: If this method returns true the control goes to controller class handler method otherwise handler method will not be executed.

  2. **default void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable ModelAndView modelAndView) **throws** Exception
     - Executes after completely executing the handler method of controller class.
     - Use case: to expose additional model objects to the view components via the given ModelAndView object like carrying system date, time and etc.

  3. **default void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, @Nullable Exception ex) **throws** Exception
     - Executes after rendering view component completely to browser.
     - Use case: allows for proper resource cleanup like nullifying request attributes, session attributes and etc.

- Every HandlerInterceptor must be registered with Spring Boot app by taking the support of InterceptorRegistry object.



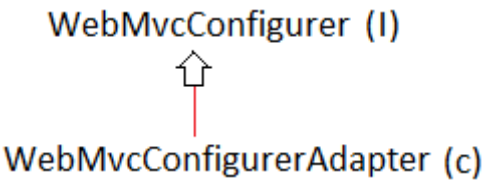
Note:

✓ Using AOP we can add pre, post logics only on Service class methods.

- ✓ Using Filters, we can add pre, post logics only on Dispatcher Servlet logic.
- ✓ Using Interceptors, we can add pre, post, after completion logic on Controller class handler methods logics.

Procedure for implementation:

Step 1: Keep any Spring Boot application ready.
Step 2: Develop HandlerInterceptor class.
Step 3: Register the HandlerInterceptor with InterceptorRegistry my getting access to registry by developing adapter Spring bean as implementing WebMvcConfigurer interface.
Step 4: Develop timeout.jsp as error page displaying the guiding message to end user in src/main/webapp folder because this JSP will be lunched without using ViewResolver support.

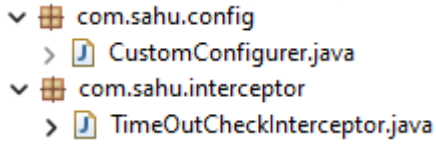


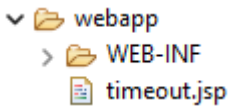
Note:

- ✓ Adapter class is class that implements interface and provides Null method definitions for Java methods so, our java class extending from Adapter class can override only those methods in which they are interested.
- ✓ After arrival of Java 8 interfaces supporting default methods there is no need adapter classes.
- ✓ WebMvcConfigurer (I) is normal Java interface up to spring 4.
- ✓ WebMvcConfigurer (I) is Java 8 interface with default methods from Spring 5, so there is no need of working with its adapter class called WebMvcConfigurerAdapter from Spring 5.

Directory Structure of BootMVCProj14-WishMessageApp-HandlerInterceptor:

- Copy and paste the BootMVCProj02-WishMessageApp and rename to BootMVCProj14-WishMessageApp-HandlerInterceptor.
- After that change the Web Project Setting context root to BootMVCProj14-WishMessageApp-HandlerInterceptor.
- Then change in <artifactId> & <name> tag of pom.xml.
- Add the following package, class and file in the project.





- Then place the following code with in their respective files.

TimeOutCheckInterceptor.java

```
package com.sahu.interceptor;

import java.time.LocalTime;

import javax.servlet.RequestDispatcher;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.HandlerInterceptor;

public class TimeOutCheckInterceptor implements HandlerInterceptor {

	public TimeOutCheckInterceptor() {
	System.out.println("TimeOutCheckInterceptor.TimeOutCheckIntercep
tor()");
	}

	@Override
	public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler)
			throws Exception {
		System.out.println("TimeOutCheckInterceptor.preHandle()");
		//Check the timing
		LocalTime localTime = LocalTime.now();
		//Get current hour of the day
		int hour = localTime.getHour();
		if (hour<9 || hour>17) {
			RequestDispatcher dispatcher =
request.getRequestDispatcher("/timeout.jsp");
			dispatcher.forward(request, response);
			return false;
		}
		return true;
	}

}
```

CustomConfigurer.java

```
package com.sahu.config;

import org.springframework.stereotype.Component;
import
org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

import com.sahu.interceptor.TimeOutCheckInterceptor;

@Component
public class CustomConfigurer implements WebMvcConfigurer {

	public CustomConfigurer() {
		System.out.println("CustomConfigurer.CustomConfigurer()");
	}

	@Override
	public void addInterceptors(InterceptorRegistry registry) {
		System.out.println("CustomConfigurer.addInterceptors()");
		registry.addInterceptor(new TimeOutCheckInterceptor());
	}

}
```

timeout.jsp

```
<%@ page isELIgnored="false"%>

<h1 style="color: red; text-align: center;">Request must be given between
9:00 a.m to 5:00 p.m</h1>
```

I18N (Internationalization) using Spring Boot MVC

- The process of making our application is working for different Locales is called applying I18N on the application.
- Locale means language + country
  E.g.,
  - en-US (English as it speaks in US)
  - fr-FR (French as it speaks in France)

- de-DE (German as it speaks in Germany)
  and etc.

- Spring Boot MVC gives built-in support for I18N.
- While enabling I18N on the application, we need to consider the following things
  a. Changing Presentation labels according to chosen Locale (main)
  b. Changing Date formats according to chosen Locale
  c. Changing Time formats according to chosen Locale
  d. Changing Number formats according to the chosen Locale
  e. Changing Currency symbols according to the chosen Locale
  f. Changing Indentation of the content (Hindi/ English/....: left to right and Urdu/ Arabic/ ….: right to left).

Procedure to enable I18N in Spring Boot MVC application

Step 1: Create Spring Boot MVC app adding the basic starters.

- Spring web
- Lombok
- JSTL (add externally)
- Tomcat Jasper (add externally)

Step 2: Create multiple Locale specific properties files having same keys and different values collected from Google translator. Like,

- App.properties (base properties file, generally contains English labels)
- App_fr_FR.properties
- App_de_DE.properties
- App_hi_IN.properties
- App_zh_CN.properties
- App_te_IN.properties

Locale specific properties files

Note: These base file name should reflect in all the Locale specific properties file names (i.e., App).

Step 3: Configure Properties file using "ResourceBundleMessageSource" class as Spring bean just specifying base file name.

Step 4: Configure SessionLocaleResolver as Spring bean as to activate I18N in Spring Boot MVC application.

Step 5: Configure "LocaleChangeInterceptor" as Spring bean to change the

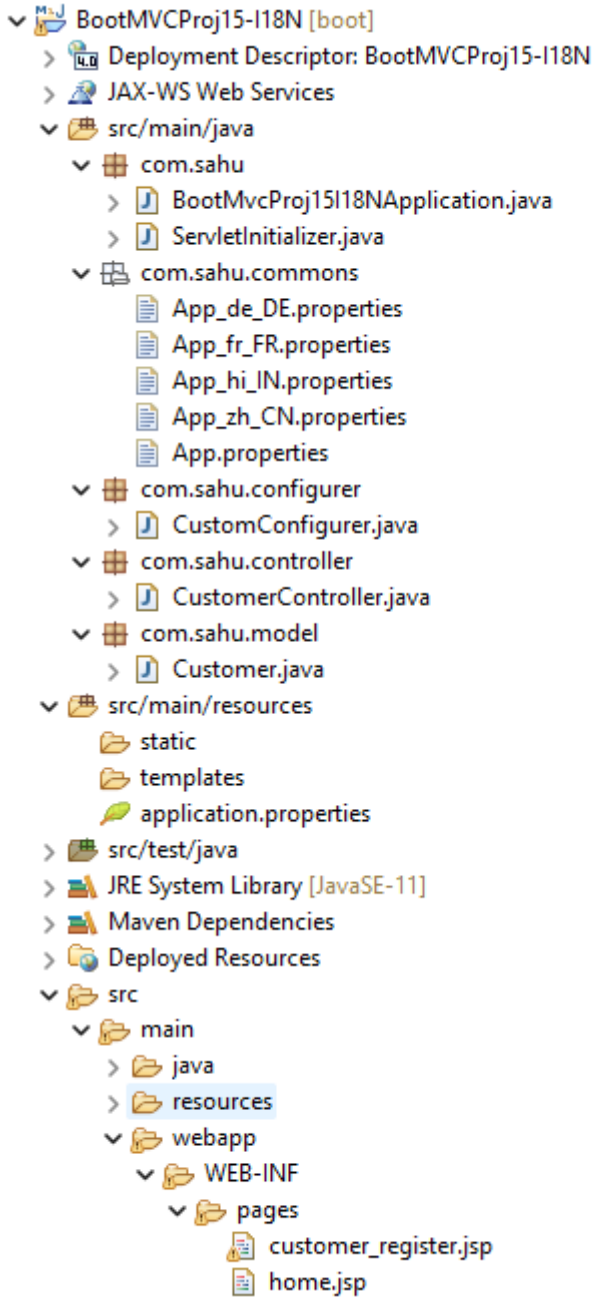
locale for each request using the locale value given by the specified request parameter.

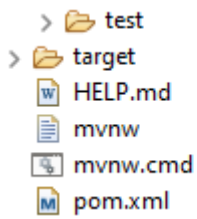
Step 6: Register the above interceptor with InterceptorRegistry.

Step 7: Develop Controller class having ability to show home page having hyperlink and form page for the hyperlink generated request.

Step 8: Develop the home page, form page reading the messages from activated Locale specific properties file using <spring:message> tag.

Directory Structure of BootMVCProj15-I18N:





- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also.
- Use following starters during project creation.
  - Spring web & Lombok
- Add the Tomcat embedded jasper jar (for embedded tomcat), JSTL jar dependency from MVN repository in pom.xml.
- Collect required properties for application.properties file from other project.
- Then place the following code with in their respective files.

App.properties

```
#Base Properties file (English)
cust.reg.title=Customer Registration Page
cust.reg.name=Customer Name
cust.reg.address=Customer Address
cust.reg.billAmount=Customer Bill Amount
cust.reg.submit=Register Customer
```

App_fr_FR.properties

```
#French Locale Properties file
cust.reg.title=Page d'enregistrement client
cust.reg.name=Nom du client
cust.reg.address=Adresse du client
cust.reg.billAmount=Montant de la facture client
cust.reg.submit=Enregistrer le client
```

App_de_DE.properties

```
#German Locale Properties file
cust.reg.title=Kundenregistrierungsseite
cust.reg.name=Kundenname
cust.reg.address=Kundenadresse
cust.reg.billAmount=Rechnungsbetrag des Kunden
cust.reg.submit=Kunde registrieren
```

App_hi_IN.properties

```
#Hindi Locale Properties file
cust.reg.title=\u0917\u094D\u0930\u093E\u0939\u0915
\u092A\u0902\u091C\u0940\u0915\u0930\u0923
\u092A\u0943\u0937\u094D\u0920
cust.reg.name=\u0917\u094D\u0930\u093E\u0939\u0915 \u0915\u093E
\u0928\u093E\u092E
cust.reg.address=\u0917\u094D\u0930\u093E\u0939\u0915 \u0915\u093E
\u092A\u0924\u093E
cust.reg.billAmount=\u0917\u094D\u0930\u093E\u0939\u0915
\u092C\u093F\u0932 \u0930\u093E\u0936\u093F
cust.reg.submit=\u0917\u094D\u0930\u093E\u0939\u0915
\u092A\u0902\u091C\u0940\u0915\u0943\u0924
\u0915\u0930\u0947\u0902
```

App_zh_CN.properties

```
#China Locale Properties file
cust.reg.title=\u5BA2\u6236\u8A3B\u518A\u9801\u9762
cust.reg.name=\u9867\u5BA2\u59D3\u540D
cust.reg.address=\u5BA2\u6236\u5730\u5740
cust.reg.billAmount=\u5BA2\u6236\u8CEC\u55AE\u91D1\u984D
cust.reg.submit=\u8A3B\u518A\u5BA2\u6236
```

BootMvcProj15I18NApplication.java

```
package com.sahu;

import java.util.Locale;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import
org.springframework.context.support.ResourceBundleMessageSource;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
import org.springframework.web.servlet.i18n.SessionLocaleResolver;

@SpringBootApplication
public class BootMvcProj15I18NApplication {
```

```
	@Bean("messageSource") //Fixed bean Id
	public ResourceBundleMessageSource createRBMS() {
		ResourceBundleMessageSource source = new
ResourceBundleMessageSource();
		source.setBasename("com/sahu/commons/App");
		return source;
	}

	@Bean("localeResolver") //Fixed bean Id
	public SessionLocaleResolver createSLResover() {
		SessionLocaleResolver resolver = new SessionLocaleResolver();
		resolver.setDefaultLocale(Locale.ENGLISH);
		return resolver;
	}

	@Bean("lci") //Not fixed bean Id
	public LocaleChangeInterceptor createLCInterceptor() {
		LocaleChangeInterceptor interceptor = new
LocaleChangeInterceptor();
		interceptor.setParamName("lang");
		return interceptor;
	}

	public static void main(String[] args) {
		SpringApplication.run(BootMvcProj15I18NApplication.class,
args);
	}

}
```

CustomConfigurer.java

```
package com.sahu.configurer;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import
org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;
```

```
@Component
public class CustomConfigurer implements WebMvcConfigurer {

	@Autowired
	private LocaleChangeInterceptor interceptor;

	@Override
	public void addInterceptors(InterceptorRegistry registry) {
		registry.addInterceptor(interceptor);
	}

}
```

Custom.java

```
package com.sahu.model;

import lombok.Data;

@Data
public class Customer {
	private String cname;
	private String caddress;
	private Double billAmount;
}
```

CustomController.java

```
package com.sahu.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;

import com.sahu.model.Customer;

@Controller
public class CustomerController {

	@GetMapping("/")
	public String showHome() {
```

```
		return "home";
	}

	@GetMapping("/customer_register")
	public String showCustomerFormPage(@ModelAttribute("customer")
Customer customer) {
		return "customer_register";
	}

}
```

home.jsp

```
<%@ page isELIgnored="false" %>

<h1 style="text-align: center;">
	<a href="customer_register">Customer Registration Page</a>
</h1>
```

customer_register.jsp

```
<%@ page isELIgnored="false" contentType="text/html; charset=UTF-8" %>
<%@ taglib uri="http://www.springframework.org/tags/form"
prefix="form" %>
<%@ taglib uri="http://www.springframework.org/tags" prefix="spring" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>

<h1 style="color: green; text-align: center;">
	<spring:message code="cust.reg.title"/>
</h1>

<form:form modelAttribute="customer">
	<table align="center" bgcolor="cyan">
		<tr>
			<td><spring:message code="cust.reg.name"/></td>
			<td><form:input path="cname"/></td>
		</tr>
		<tr>
			<td><spring:message code="cust.reg.address"/></td>
			<td><form:input path="caddress"/></td>
```

```
		</tr>
		<tr>
			<td><spring:message code="cust.reg.billAmount"/></td>
			<td><form:input path="billAmount"/></td>
		</tr>
		<tr>
			<td colspan="2">
				<input type="submit" value="<spring:message
code="cust.reg.submit"/>">
			</td>
		</tr>
	</table>
</form:form>
<br>

<div style="text-align: center;">
	<fmt:setLocale value="${response.locale}"/>
	<jsp:useBean id="dt" class="java.util.Date"/>
	<fmt:formatDate value="${dt}" var="fDate" type="date"
dateStyle="FULL"/>
	System Date : ${fDate}<br>
	<fmt:formatDate value="${dt}" var="fTime" type="time"/>
	System Date : ${fTime}<br>

	<fmt:formatNumber value="10000000000" var="fPrice"
type="currency"/>
	Item Price : ${fPrice}<br>
	<fmt:formatNumber value="9876543210" var="fNumber"
type="number"/>
	Item Number : ${fNumber}<br>
</div>

<br>
<p style="text-align: center;">
	<a href="?lang=en_US">English</a>  
	<a href="?lang=fr_FR">French</a>  
	<a href="?lang=de_DE">German</a>  
	<a href="?lang=hi_IN">Hindi</a>  
	<a href="?lang=zh_CN">Chainese</a>  
</p>
```

I18N application Flow

a. Pre instantiation of Spring bean

BootMvcProj15I18NApplication.java

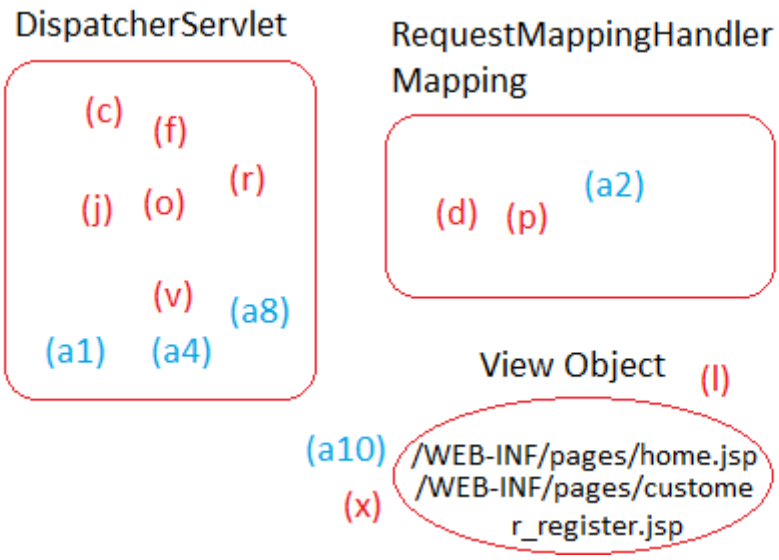
```
@SpringBootApplication
public class BootMvcProj15I18NApplication {
(a)     @Bean("messageSource") //Fixed bean Id
        public ResourceBundleMessageSource createRBMS() {
                ResourceBundleMessageSource source = new
ResourceBundleMessageSource();
                source.setBasename("com/sahu/commons/App");
                return source;
        }
(a)     @Bean("localeResolver") //Fixed bean Id
        public SessionLocaleResolver createSLResover() {
                SessionLocaleResolver resolver = new SessionLocaleResolver();
                resolver.setDefaultLocale(Locale.ENGLISH);
                return resolver;
        }
                since no “lang” request param is coming along with the request it
                will fall back to default Locale App_en_US.propeties file is not
                available it will activate the base properties file App.properteis.
        @Bean("lci") //Not fixed bean Id        (g)     (s) same is (g)
(a)     public LocaleChangeInterceptor createLCInterceptor() {
                LocaleChangeInterceptor interceptor = new
LocaleChangeInterceptor();      (a5) since the request is having lang=”hi_IN”, it
                interceptor.setParamName("lang");    will pick up the
                return interceptor;         App_hi_IN.properties file as the current
        }                                           Locale specific property file
        public static void main(String[] args) {
                SpringApplication.run(BootMvcProj15I18NApplication.class, args);
        }
}
```

CustomConfigurer.java

```
@Component      (a)
public class CustomConfigurer implements WebMvcConfigurer {
        @Autowired
        private LocaleChangeInterceptor interceptor;
```

```
	@Override
	public void addInterceptors(InterceptorRegistry registry) {
		registry.addInterceptor(interceptor);
	}
}
```

(b) *http://localhost:2525/BootMVCProj15-I18N/*



CustomController.java

```
@Controller
public class CustomerController {
	@GetMapping("/")
(h)	public String showHome() {	(e?)
		return "home"; (i)
	}
(t)	@GetMapping("/customer_register") (q?)	(a3?)
(a6)	public String showCustomerFormPage(@ModelAttribute("customer")
Customer customer) {
		return "customer_register";	(u)	(a7)
	}
}
```

application.properties

```
#ViewResolver	(k)	(w)	(a9)
spring.mvc.view.prefix=/WEB-INF/pages/
spring.mvc.view.suffix=.jsp
```

home.jsp (m)

```
<%@ page isELIgnored="false" %>
```

```
<h1 style="text-align: center;">
        <a href="customer_register">Customer Registration Page</a>  (n)
</h1>
```

Displays the labels for given keys/ codes by collecting form App_hi_IN.properties file.

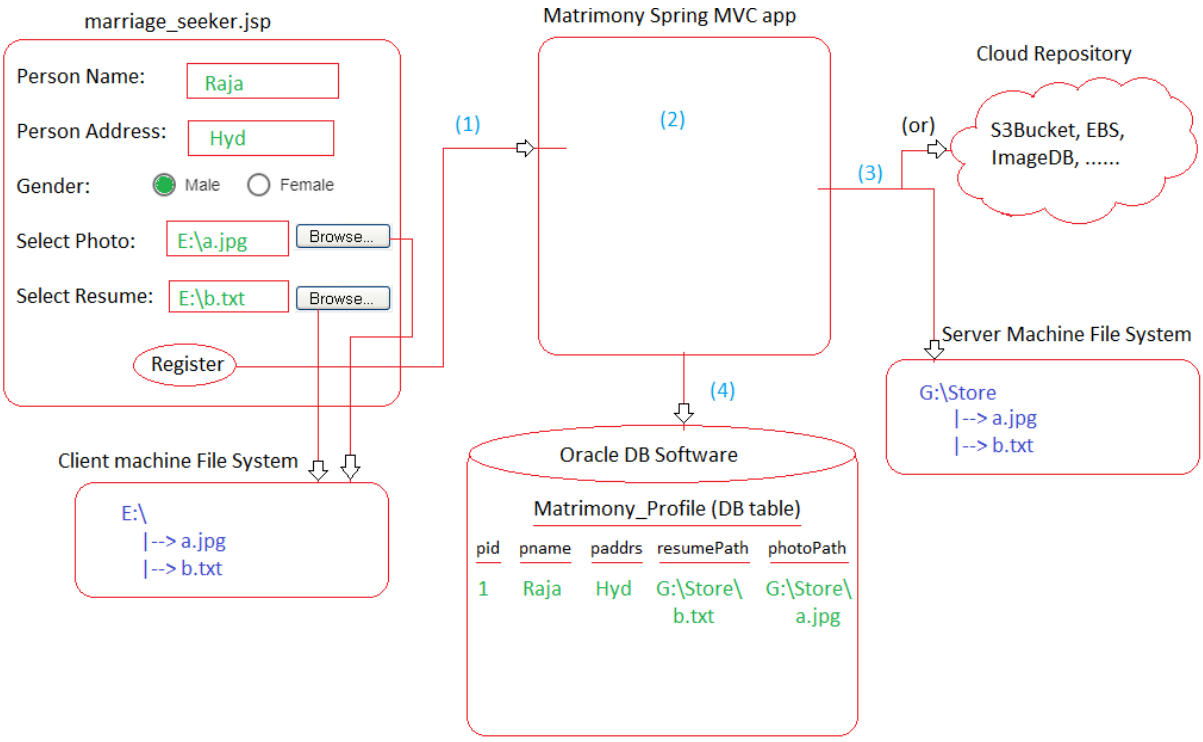
Displays the labels by collecting the values for the given code/ keys specified in App.properties (base file) [English labels]

Customer_register.jsp (y)

```
<h1 style="color: green; text-align: center;">
        <spring:message code="cust.reg.title"/>
</h1>                                   (y)     (a11)
<form:form modelAttribute="customer">
        <table align="center" bgcolor="cyan">
                <tr>
                        <td><spring:message code="cust.reg.name"/></td>
                        <td><form:input path="cname"/></td>
                </tr>
                <tr>                                                (y)     (a11)
                        <td><spring:message code="cust.reg.address"/></td>
                        <td><form:input path="caddress"/></td>
                </tr>
                <tr>                                                (y)     (a11)
                        <td><spring:message code="cust.reg.billAmount"/></td>
                        <td><form:input path="billAmount"/></td>
                </tr>
                <tr>
                        <td colspan="2">
                                <input type="submit" value="<spring:message
code="cust.reg.submit"/>">
        (y)     (a11)  </td>
                </tr>
        </table>
</form:form> <br>
<p style="text-align: center;">
        <a href="?lang=en_US">English</a>  
        <a href="?lang=fr_FR">French</a>  
        <a href="?lang=de_DE">German</a>  
        <a href="?lang=hi_IN">Hindi</a>       (z)
        <a href="?lang=zh_CN">Chainese</a>  
</p>
```

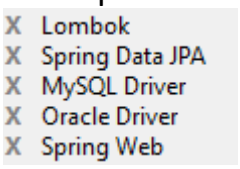
File Uploading and File Downloading

- The process of selecting files from client machine file system and sending them to server machine file system is called File uploading and reverse is called File downloading.
- The matrimony apps, Job portal apps, Profile management apps and etc. need file uploading and downloading activities.
- Generally, we don't store uploaded files in the DB s/w as BLOB or CLOB column values, we store them in sever machine file system but we add their address paths to DB s/w as String values.



Procedure to perform File Uploading in Spring Boot MVC application

Step 1: Create Spring Boot starter Project of type war adding the following staters and dependencies



And add the following jars in pom.xml by collecting form MVN repository.

- Tomcat embedded jasper
- commons fileupload
- commons- io
- JSTL

Step 2: Develop Entity class, Repository interface, Service interface and service implementation class.

Step 3: Develop Model class to hold form data having “MultipartFile” type properties to hold the uploaded content (Internally streams).

Note: MultipartFile is internally streams representing the uploaded files that are stored in the InMemory of RAM or TEMP folder of HDD.

Step 4: Develop controller class handler method having logic to display home page for “/” request path.

Step 5: Develop Handler method in controller class showing the form page having file upload components and also the register form page.

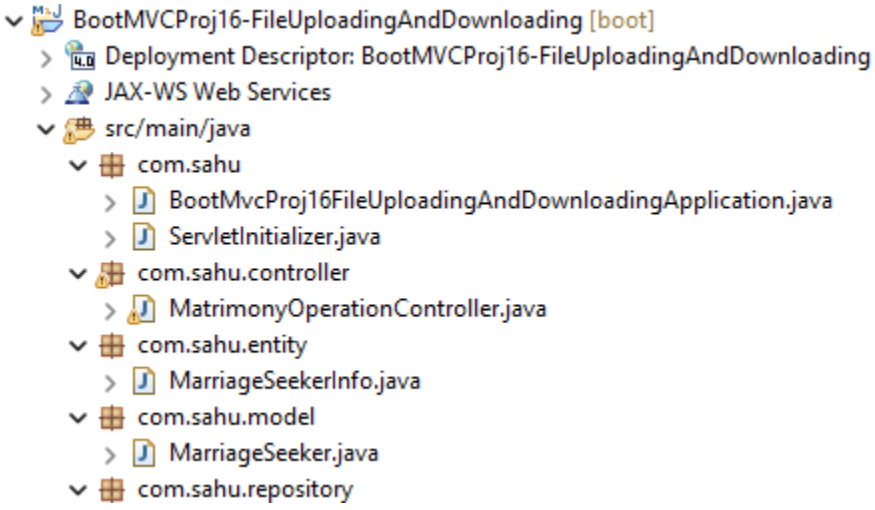
Note: enctype=*"multipart/form-data"* this makes server to receive non text data long with the request.

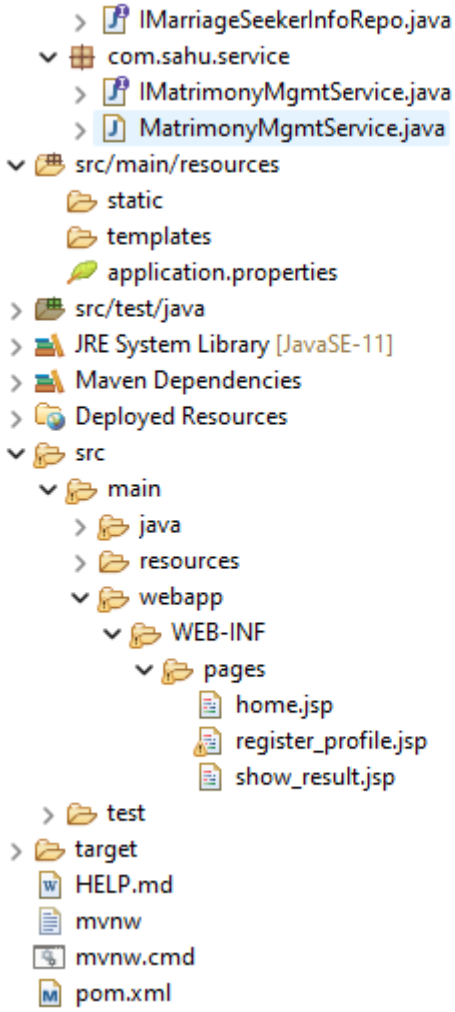
Step 6: Configure “CommonsMultipartResolver” as Spring bean in @SpringBootApplicaiton class to alert Spring Boot MVC application to receive and bind uploaded files content to multipart file type properties.

Step 7: Develop another handler method in controller class to process post mode request of form submission having logic to complete file uploading activity.

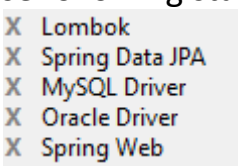
Step 8: Develop rest thing and run the application.

Directory Structure of BootMVCProj15-I18N:





- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also.
- Use following starters during project creation.



- Add the Tomcat embedded jasper (for embedded tomcat), JSTL, commons fileupload, commons- io jars dependency in pom.xml from MVN repository.
- Then place the following code with in their respective files.

application.properties

```
#ViewResolver
spring.mvc.view.prefix=/WEB-INF/pages/
spring.mvc.view.suffix=.jsp
```

```
#Embedded Server port number
server.port=4547

#Context path for standalone execution
server.servlet.context-path=/FileUpoadDownload

#Connection provider to work HikariCP
spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver
spring.datasource.url=jdbc:oracle:thin:@localhost:1521:xe
spring.datasource.username=system
spring.datasource.password=manager
spring.datasource.hikari.minimum-idle=10
spring.datasource.hikari.maximum-pool-size=100
spring.datasource.hikari.idle-timeout=60000

#Spring Data JPA Hibernate properties
spring.jpa.database-platform=org.hibernate.dialect.Oracle10gDialect
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto=update

#Location in server machine file to store upload files
upload.store=C:/Users/Nirmala/Downloads/seekes_info
```

MarriageSeekerInfo.java

```
package com.sahu.entity;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

import lombok.Data;

@Data
@Entity
@Table(name = "BOOT_MARRIAGE_SEEKER_INFO")
public class MarriageSeekerInfo {
	@Id
	@GeneratedValue
```

```
        private Integer profileId;
        @Column(length = 20)
        private String ProfileName;
        @Column(length = 10)
        private String gender;
        @Column(length = 200)
        private String resumePath;
        @Column(length = 200)
        private String photoPath;
}
```

MarriageSeeker.java

```
package com.sahu.model;

import org.springframework.web.multipart.MultipartFile;

import lombok.Data;

@Data
public class MarriageSeeker {
        private String profileName;
        private String gender="female";
        private MultipartFile resume;
        private MultipartFile photo;
}
```

BootMvcProj16FileUploadingAndDownloadingApplication.java

```
@SpringBootApplication
public class BootMvcProj16FileUploadingAndDownloadingApplication {

        @Bean("multpartResolver")
        public CommonsMultipartResolver createCMResolver() {
                CommonsMultipartResolver  resolver = new
CommonsMultipartResolver();
                resolver.setMaxUploadSize(-1);
                resolver.setMaxUploadSizePerFile(20*1024*1024);
                resolver.setPreserveFilename(true);
                return resolver;
        }
```

IMarriageSeekerInfoRepo.java

```
package com.sahu.repository;

import org.springframework.data.repository.PagingAndSortingRepository;

import com.sahu.entity.MarriageSeekerInfo;

public interface IMarriageSeekerInfoRepo extends
PagingAndSortingRepository<MarriageSeekerInfo, Integer> {

}
```

IMatrimonyMgmtService.java

```
package com.sahu.service;

import com.sahu.entity.MarriageSeekerInfo;

public interface IMatrimonyMgmtService {
	public String registerProfile(MarriageSeekerInfo seekerInfo);
}
```

MatrimonyMgmtService.java

```
package com.sahu.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.entity.MarriageSeekerInfo;
import com.sahu.repository.IMarriageSeekerInfoRepo;

@Service("matrimonyService")
public class MatrimonyMgmtService implements IMatrimonyMgmtService {

	@Autowired
	private IMarriageSeekerInfoRepo marriageSeekerInfoRepo;

	@Override
	public String registerProfile(MarriageSeekerInfo seekerInfo) {
		return
```

```
marriageSeekerInfoRepo.save(seekerInfo).getProfileName()+" details has
registered";
        }

}
```

MatrimonyOperationController.java

```
package com.sahu.controller;

import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.Map;

import org.apache.commons.io.IOUtils;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.core.env.Environment;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.multipart.MultipartFile;

import com.sahu.entity.MarriageSeekerInfo;
import com.sahu.model.MarriageSeeker;
import com.sahu.service.IMatrimonyMgmtService;

@Controller
public class MatrimonyOperationController {

        @Autowired
        private IMatrimonyMgmtService matrimonyMgmtService;

        @Autowired
        private Environment environment;

        @GetMapping("/")
        public String showHome() {
                return "home";
```

```
	}

	@GetMapping("/register")
	public String showRegisterPage(@ModelAttribute("seeker")
MarriageSeeker seeker) {
		return "register_profile";
	}

	@PostMapping("/register")
	public String registerMarriageSeeker(@ModelAttribute("seeker")
MarriageSeeker seeker, Map<String, Object> map) throws Exception {
		//Get the folder location to store uploaded files
		String location = environment.getProperty("upload.store");
		File locationStore = new File(location);
		//Crate Location folder if it is not already available
		if(!locationStore.exists())
			locationStore.mkdir();
		//Get original names of the uploaded files
		MultipartFile resumeFile = seeker.getResume();
		MultipartFile photoFile = seeker.getPhoto();
		String resumeFileName = resumeFile.getOriginalFilename();
		String photoFileName = photoFile.getOriginalFilename();
		//Create InputStreams representing the Uploaded files
		InputStream resumeIStream = resumeFile.getInputStream();
		InputStream photoIStream = photoFile.getInputStream();
		//Create OutputStreams pointing destination file o the server
machine file system
		OutputStream resumeOStream = new
FileOutputStream(location+"/"+resumeFileName);
		OutputStream photoOStream = new
FileOutputStream(location+"/"+photoFileName);
		//Complete copy operation among the streams
		IOUtils.copy(resumeIStream, resumeOStream);
		IOUtils.copy(photoIStream, photoOStream);
		//Close Streams
		resumeIStream.close(); resumeOStream.close();
		photoIStream.close(); photoOStream.close();
		//Create Entity class object
		MarriageSeekerInfo seekerInfo = new MarriageSeekerInfo();
		seekerInfo.setProfileName(seeker.getProfileName());
```

```
                seekerInfo.setGender(seeker.getGender());
                seekerInfo.setResumePath(location+"/"+resumeFileName);
                seekerInfo.setPhotoPath(location+"/"+photoFileName);
                //Use service
                String message =
matrimonyMgmtService.registerProfile(seekerInfo);
                //Create model attributes
                map.put("resumeFile", resumeFileName);
                map.put("photoFile", photoFileName);
                map.put("resultMsg", message);
                //return LVN
                return "show_result";
        }

}
```

home.jsp

```
<%@ page isELIgnored="false" %>

<h1 style="text-align: center;">
        <a href="register">Register Profile</a>
</h1>
<br>
<h1 style="text-align: center;">
        <a href="display">Display Profile</a>
</h1>
```

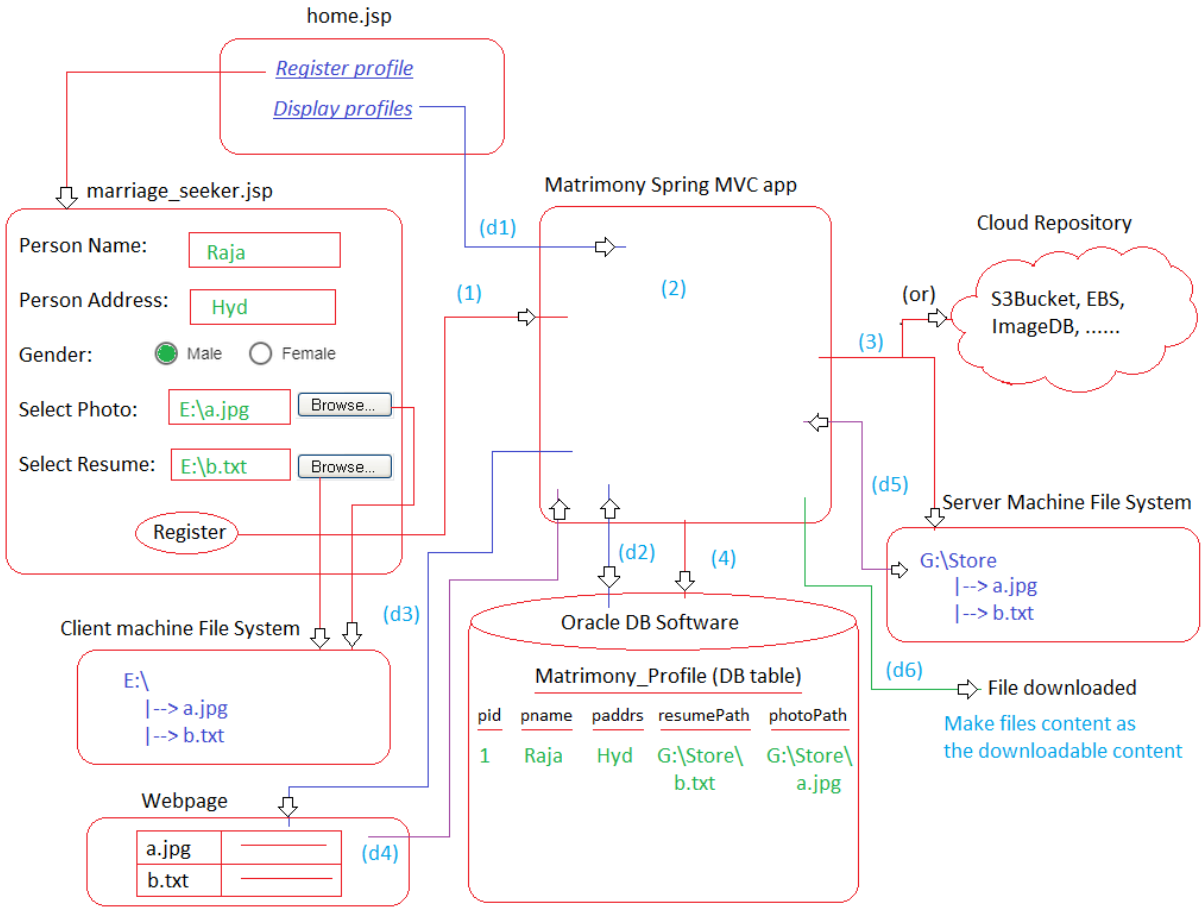
register_profile.jsp

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

<h1 style="color: red; text-align: center;">Marriage Seeker
Registration</h1>

<form:form modelAttribute="seeker" enctype="multipart/form-data">
        <table align="center" bgcolor="cyan">
                <tr>
                        <td>Marriage Seeker Name:</td>
```

```
                <td><form:input path="profileName"/></td>
            </tr>
            <tr>
                <td>Marriage Seeker Gender:</td>
                <td>
                    <form:radiobutton path="gender"
value="male"/>Male   
                    <form:radiobutton path="gender"
value="female"/>Female
                </td>
            </tr>
            <tr>
                <td>Select Resume:</td>
                <td><form:input type="file" path="resume"/></td>
            </tr>
            <tr>
                <td>Select Photo:</td>
                <td><form:input type="file" path="photo"/></td>
            </tr>
            <tr>
                <td></td>
                <td>
                    <input type="reset" value="Cancel">
  
                    <input type="submit" value="Register">
                </td>
            </tr>
        </table>
</form:form>
```

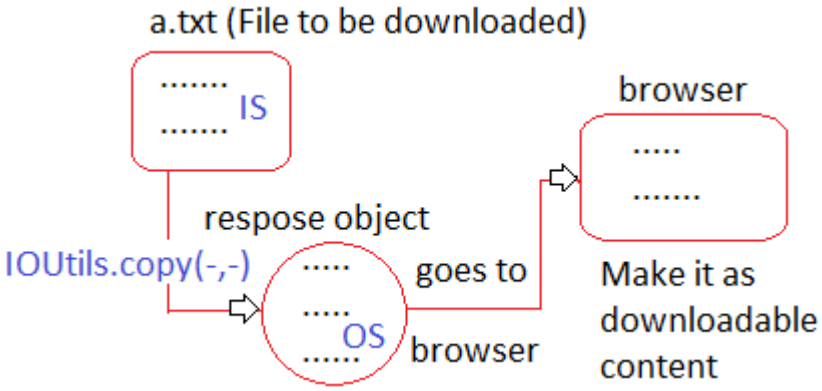
show_result.jsp

```
<%@ page isELIgnored="false" %>

<h1 style="color: red; text-align: center;">Result Page</h1>
<h3 style="text-align: center;">Uploaded File names are: ${resumeFile},
${photoFile}</h3>
<h3 style="text-align: center; color: green;">${resultMsg}</h3>
<br>
<a href="./">Home</a>
```



Standard procedure for file downloading

a. Receive the name and location of file to be downloaded to the handler method from additional request param value of the request using @RequestParam annotation.
b. Get the length of the file to be downloaded and make it as the response length.
c. Get the MIME type of the file to be downloaded and make it as the response content type.
d. Create InputStream pointing to the file to be downloaded.
e. Create OutputStream pointing to the Response object.



f. Give instruction to browser using response header "Content-Disposition" to make the received content as the downloadable file.
g. Copy the content of file to be downloaded to response object using streams with the help of IOUtils.copy(-,-) method.

- Create a show_profiles.jsp under webapp/WEB-INF/pages and place the following code in their respective files.

IMatrimonyMgmtService.java

```
public Iterable<MarriageSeekerInfo> getAllProfiles();
```

MatrimonyMgmtService.java

```
@Override
public Iterable<MarriageSeekerInfo> getAllProfiles() {
	return marriageSeekerInfoRepo.findAll();
}
```

MatrimonyOperationController.java

```
@Autowired
private ServletContext context;

@GetMapping("/display")
public String displayProfile(Map<String, Object> map){
	Iterable<MarriageSeekerInfo> seekerList =
matrimonyMgmtService.getAllProfiles();
	map.put("seekersInfo", seekerList);
	//return LVN
	return "show_profiles";
}

@GetMapping("/download")
public String fileDownload(@RequestParam("file") String filePath,
HttpServletResponse response) throws Exception {
	//create java.io.File object pointing to the file to be download
	File file = new File(filePath);
	//Get length of the file and make it as the response content
length
	response.setContentLengthLong(file.length());
	//Get the MIME type of the file to be download and make it as
```

```
the response content type
                String mimeType = context.getMimeType(filePath);
                mimeType= mimeType!=null?mimeType:"application/octet-
stream";
                response.setContentType(mimeType);
                //Create InputStream pointing to the file to be downloaded
                InputStream inputStream = new FileInputStream(filePath);
                //Create OutputStream pointing to the response object
                OutputStream outputStream=response.getOutputStream();
                //give instruction to browser to make the received content as
the download file
                response.setHeader("content-Disposition",
"attachment;fileName="+file.getName());
                //Copy file to be download content to response object.
                IOUtils.copy(inputStream, outputStream);
                //close the streams
                inputStream.close();
                outputStream.close();
                //This indicate  respose should go to browser directly with out
taking the support of ViewResolver and View components.
                return null;
        }
```

show_profiles.jsp

```
<%@ page isELIgnored="false" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>

<c:choose>
        <c:when test="${!empty seekersInfo}">
                <table border="1" align="center" bgcolor="pink">
                        <tr bgcolor="yellow">
                                <th>Profile ID</th>
                                <th>Profile Name</th>
                                <th>Gender</th>
                                <th>Resume</th>
                                <th>Photo</th>
                        </tr>
                        <c:forEach var="seeker" items="${seekersInfo}">
                                <tr>
```

```
                        <tr>
                            <td>${seeker.profileId}</td>
                            <td>${seeker.profileName}</td>
                            <td>${seeker.gender}</td>
                            <td>
                                <a
href="download?file=${seeker.resumePath}">${fn:substringAfter(seeker.res
umePath, 'C:/Users/Nirmala/Downloads/')}</a>
                            </td>
                            <td>
                                <a
href="download?file=${seeker.photoPath}">${fn:substringAfter(seeker.phot
oPath, 'C:/Users/Nirmala/Downloads/')}</a>
                            </td>
                        </tr>
                    </c:forEach>
                </table>
            </c:when>
            <c:otherwise>
                <h1 style="color: red; text-align: center;">Records not
found</h1>
            </c:otherwise>
</c:choose>
```

ViewResolver in Spring Boot MVC

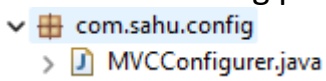
- These are given to map the given LVN with physical View component and return View object having the name and location physical View component.
- All View resolvers are pre-defined classes implementing org.springframework.web.servlet.ViewResolver (I)
  E.g.,
  - InternalResourceViewResolver
  - UrlBasedViewResolver
  - ResourceBundleViewResolver
  - XmlViewResolver
  - TilesViewResolver
  - BeanNameViewResolver and etc.

Note: We generally use one of these pre-defined ViewResolver and we don't need and custom ViewResolver.

- Every ViewResolver must be registered with ViewResolverRegistry by getting the registry through WebMvcConfigurer class development (or) by writing entries in application.properties.

Directory Structure of BootMVCProj17-WishMessageApp-ViewResolvers:

- Copy and paste the BootMVCProj02-WishMessageAppand rename to BootMVCProj17-WishMessageApp-ViewResolvers.
- After that change the Web Project Setting context root to BootMVCProj17-WishMessageApp-ViewResolvers.
- Then change in <artifactId> & <name> tag of pom.xml.
- Add the following package and class under src/main/java.



- Comment ViewResolver related entries in application.properties.
- Add the following package, class and file in the project.

MVCConfigurer.java

```
package com.sahu.config;

import org.springframework.stereotype.Component;
import
org.springframework.web.servlet.config.annotation.ViewResolverRegistry;
import
org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import
org.springframework.web.servlet.view.InternalResourceViewResolver;

@Component
public class MVCConfigurer implements WebMvcConfigurer {

	@Override
	public void configureViewResolvers(ViewResolverRegistry registry) {
	System.out.println("MVCConfigurer.configureViewResolvers()");
		InternalResourceViewResolver resolver =  new
InternalResourceViewResolver();
		resolver.setPrefix("/WEB-INF/pages/");
		resolver.setSuffix(".jsp");
		registry.viewResolver(resolver);
	}

}
```

Note: If both means WebMvcConfigurer implementation class and application.properties file are given then the WebMvcConfigurer implementation class configurations should takes place.

UrlBasedViewResolver

- Allows to take any technology components as the view components including Servlet, JSP components.
- We must set View class explicitly to decide the View technology to use.
- The View classes are the classes implementing View (I) like JstlView, InternalResourceView, TilesView, FreeMarkerView and etc.
- More E.g.,

```
AbstractAtomFeedView, AbstractFeedView, AbstractJackson2View,
AbstractPdfStamperView, AbstractPdfView, AbstractRssFeedView,
AbstractTemplateView, AbstractUrlBasedView, AbstractView,
AbstractXlsView, AbstractXlsxStreamingView, AbstractXlsxView,
FreeMarkerView, GroovyMarkupView, InternalResourceView,
JstlView, MappingJackson2JsonView, MappingJackson2XmlView,
MarshallingView, RedirectView, ScriptTemplateView, TilesView,
XsltView
```

- To use Servlet, JSP components as the View components we can use either InternalResourceView (if JSTL tags are not used) or JstlView (if JSTL tags are used).
- UrlBasedViewResolver is the super class of InternalResourceViewResolver.

MVCConfigurer.java

```
@Component
public class MVCConfigurer implements WebMvcConfigurer {
	@Override
	public void configureViewResolvers(ViewResolverRegistry registry) {
	System.out.println("MVCConfigurer.configureViewResolvers()");
		UrlBasedViewResolver resolver = new UrlBasedViewResolver();
		resolver.setPrefix("/WEB-INF/pages/");
		resolver.setSuffix(".jsp");
		resolver.setViewClass(InternalResourceView.class);
		//resolver.setViewClass(JstlView.class)
		registry.viewResolver(resolver);
	}
}
```

Note: We can also configure ViewResolver using @Bean method of main class (@SpringBootApplication class)

@SpringBootApplication class

```
@Bean
public UrlBasedViewResolver createUBVR() {
        UrlBasedViewResolver resolver = new UrlBasedViewResolver();
        resolver.setPrefix("/WEB-INF/pages/");
        resolver.setSuffix(".jsp");
        resolver.setViewClass(InternalResourceView.class);
        return resolver;
}
```

Q. What is the difference b/w InternalResourceView and JstlView classes?

Ans.

- Both are given to use Servlet, JSP components of private area as the view components.
- If InternalResourceView is used we need to add JSTL jar files only when JSTL tags are used in the JSP pages otherwise not required.
- If JstlView is used we need to add JSTL jar files irrespective of JSTL tags are used or not in the JSP pages.

Q. What is difference b/w UrlBasedViewResolver and InternalResourceViewResolver?

Ans.

UrlBasedViewResolver	InternalResourceViewResolver
a. Allows to view components in any technology of your choice.	a. Allows to take only Servlet, JSP of private area as the view components.
b. View class configuration explicitly is mandatory, no default View class.	b. View class configuration is optional because takes InternalResourceView as the default view class if JSTL jars are not added, if added then takes JstlView as default view class.
c. Super class for InternalResourceViewResolver.	c. Sub class for UrlBasedViewResolver.
d. Must be configured and registered using ViewResolverRegistry of WebMvcConfigurer implementation	d. Can be configured and registered using ViewResolverRegistry of WebMvcConfigurer implementation class (or) using entries in

class (or) using @Bean method of main class.	application.properties file. (or) using @Bean Method of main class.

Note:

- ✓ The biggest limitation in the above both ViewResolver is the LVN given by Controller class handler method must be taken as the view component name as show below,

LVN	Physical View component
display	/WEB-INF/pages/display.jsp
home	/WEB-INF/pages/home.jsp

- ✓ Shows tight coupling between LVN and view component name.
- ✓ While working with the above two view resolvers we need to take all view components in the same location (prefix) and in the same technology (suffix).

Limitations of UrlBasedViewResolver and InternalResourceViewResolver:

a. Gives tight coupling between LVNs and physical View components.
b. All view components must be there in same technology and location.

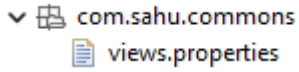
Note: To overcome these problems take the support of ResourceBundleViewResolver or XmlViewResolver.

- The actually ways of registering ViewResolver is, working with ViewResolver registry and the shortcut ways of do that is,
  - Using @Bean methods in Main class (can be used for all kinds of View Resolvers)
  - Using entries of application.properties (only for InternalResourceViewResolver)

ResourceBundleViewResolver

- Allows us to take different View components in different locations and in different technologies.
- We can get loose coupling between LVNs and physical view components.
- Needs to use separate properties file to specify the view component details.
- The default properties file name is "views.properties" of class path folder (src/main/java).

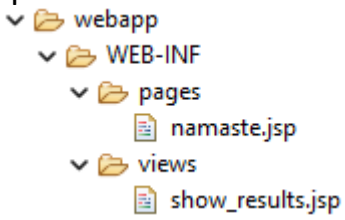
Step 1: Prepare properties file having view component details (For every LVN, we can specify separate View class and separate name, location and technology).



views.properties

```
#View configuration
home.(class)=org.springframework.web.servlet.view.InternalResourceView
home.url=/WEB-INF/pages/namaste.jsp

display.(class)=org.springframework.web.servlet.view.JstlView
home.url=/WEB-INF/views/show_results.jsp
```

- According that create the folder structure and rename the files also (home.jsp to namaste.jsp and display.jsp to show_result.jsp) and keep in proper location as shown below.



Step 2: Configure ResourceBundleViewResolver either as @Bean on using WebMvcConfigurer implementation class specifying the above properties file.

@SpringBootApplication class

```
@Bean
public ResourceBundleViewResolver createRBVR() {
        ResourceBundleViewResolver resolver = new
ResourceBundleViewResolver();
        resolver.setBasename("com/sahu/commons/views");
        return resolver;
}
```

Note:

- ✓ The com/sahu/commons package can be created either in src/main/java folder or in src/main/resources folder.
- ✓ We cannot use directly html files without thymeleaf technology as view components in Spring MVC web applications.

XmlViewResolver

- Same as ResourceBundleViewResolver but allows us to take .xml file for the view configuration (default name and location is views.xml in WEB-INF folder).
- The XML file Spring bean configuration file having "LVN" as the bean id and the View class name as bean class name injecting the name and location physical view component.

Step 1: Develop Spring bean configuration file (views.xml) in WEB-INF folder specifying the view configurations.

views.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd">
  <bean id="home"
   class="org.springframework.web.servlet.view.InternalResourceView">
    <property name="url" value="/WEB-INF/pages/namaste.jsp"/>
  </bean>
  <bean id="display"
   class="org.springframework.web.servlet.view.JstlView">
    <property name="url" value="/WEB-INF/views/show_results.jsp"/>
  </bean>
</beans>
```

Step 2: Configure XmlViewResolver using @Bean method in main class or using the support of WebMvcConfigurer implementation class.

@SpringBootApplication class

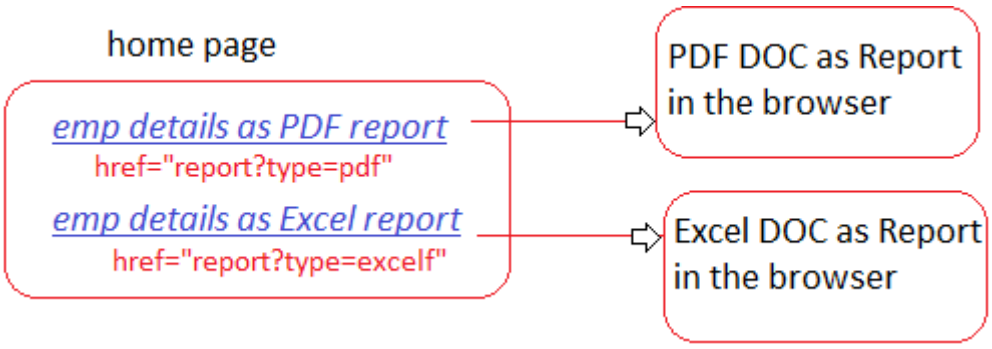
```
        @Autowired
        private ServletContext sc;
        @Bean
        public XmlViewResolver createXVR() {
                XmlViewResolver resolver = new XmlViewResolver();
                resolver.setLocation(new
FileSystemResource(sc.getRealPath("/")+"/WEB-INF/views.xml"));
                return resolver;
        }
```

Note:

- ✓ resolver.~~setLocation~~(**new** FileSystemResource(sc.getRealPath("/")+"/WEB-INF/views.xml")); this line is optional to write if the location of the XML file is WEB-INF/views.jsp.
- ✓ All these View Resolvers (UrlBasedViewResolver, InternalResourceViewResolver, ResourceBundleViewResolver, XmlViewResolver) does not support view Resolver chaining.

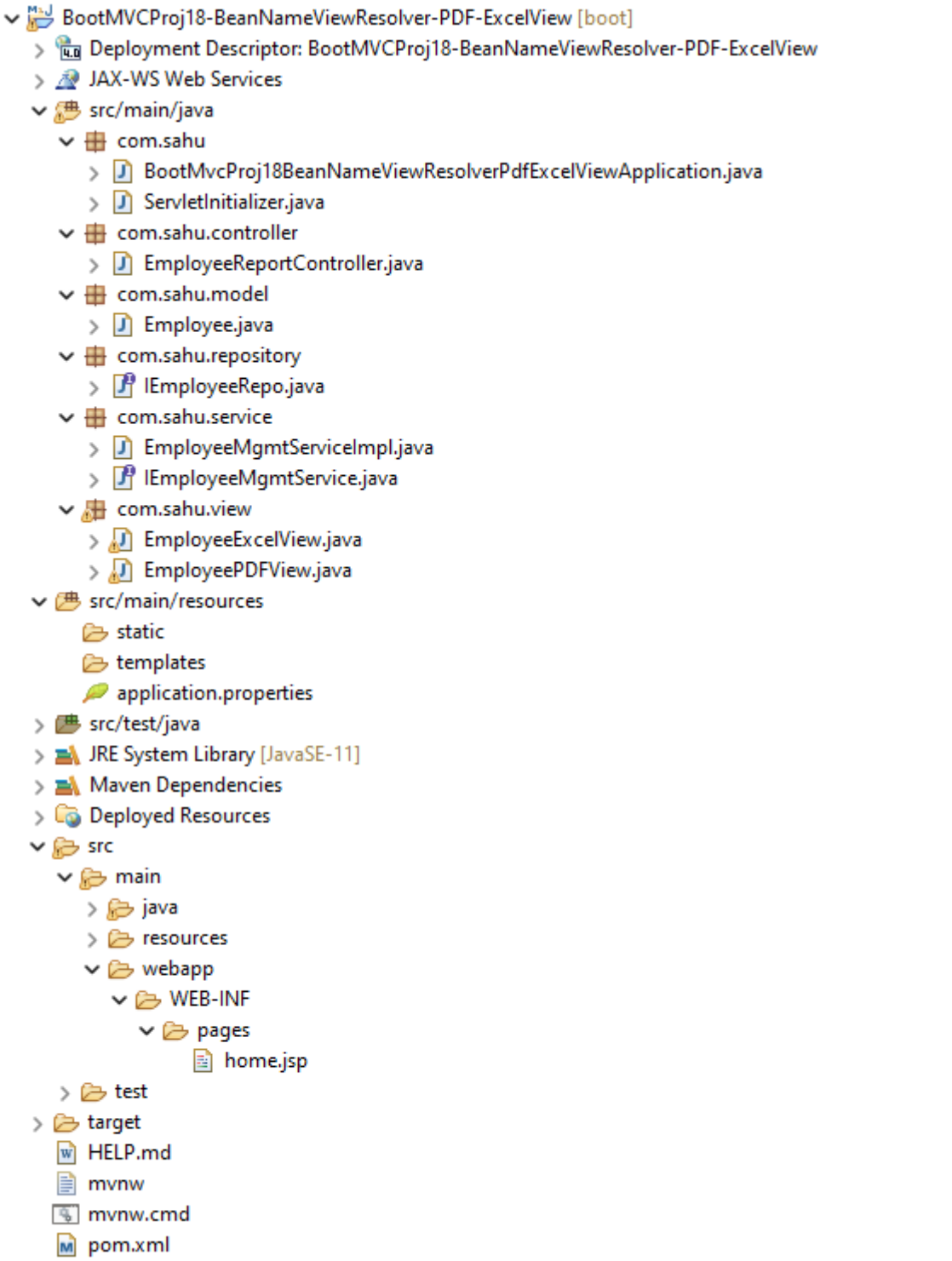
Generating Report as Excel and PDF documents

- ➢ Spring MVC or spring Boot MVC View is an abstract entity i.e., we can take anything as the view component like JSP, Freemarker, HTML with Thymeleaf, Velocity, Tiles, Java classes and etc.
- ➢ To generate reports as Excel sheet we need the support Apache POI API, for PDF report we need the support of iText API.
- ➢ To place these third-party APIs, we need to take Java classes/ Spring bean as the view components implementing View (I) directly or indirectly.
- ➢ To choose Spring beans as view component based on the received LVN we need to work with "BeanNameViewResolver" (This supports ViewResolver chaining).
- ➢ This ViewResolver maps given LVN with that View component which is Spring bean having LVN as the bean id.



- ➢ Generally, the InternalResourceView, JstlView class object acts as View object to render JSP pages as the view components but placing POI API, iText API in JSP pages is bad practice because they are Java statements.
- ➢ To overcome these problems, we have to develop our own View classes adding POI API, iText API support.
- ➢ Use the particular version of POI, iText API because of version issue.
  - poi-3.17
  - iText-2.1.7

Directory Structure of BootMVCProj18-BeanNameViewResolver-PDF-ExcelView:



- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also
- Use following starters during project creation.

- Add the Tomcat embedded jasper jar (for embedded tomcat), POI, iText jar dependency from MVN repository in pom.xml.
- Collect the application.properties from other project (Mini project) and must have view resolver related entries.
- Then place the following code with in their respective files.

Employee.java

```
package com.sahu.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

import lombok.Data;

@Entity
@Table(name="emp")
@Data
public class Employee {
	@Id
	@GeneratedValue
	private Integer empno;
	private String ename;
	private String job;
	private Double sal;
	private Integer deptno;
}
```

IEmployeeRepo.java

```
package com.sahu.repository;

import org.springframework.data.repository.PagingAndSortingRepository;

import com.sahu.model.Employee;

public interface IEmployeeRepo extends
PagingAndSortingRepository<Employee, Integer> {
}
```

IEmployeeMgmtService.java

```
package com.sahu.service;

import com.sahu.model.Employee;

public interface IEmployeeMgmtService {
	public Iterable<Employee> getAllEmployees();
}
```

EmployeeMgmtServiceImpl.java

```
package com.sahu.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.model.Employee;
import com.sahu.repository.IEmployeeRepo;

@Service("empService")
public class EmployeeMgmtServiceImpl implements
IEmployeeMgmtService {

	@Autowired
	private IEmployeeRepo employeeRepo;

	@Override
	public Iterable<Employee> getAllEmployees() {
		return employeeRepo.findAll();
	}

}
```

EmployeeReportController.java

```
package com.sahu.controller;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
```

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestParam;

import com.sahu.model.Employee;
import com.sahu.service.IEmployeeMgmtService;

@Controller
public class EmployeeReportController {

	@Autowired
	private IEmployeeMgmtService employeeMgmtService;

	@GetMapping("/")
	public String showHome() {
		return "home";
	}

	@GetMapping("/report")
	public String generateReprot(@RequestParam("type") String type,
Map<String, Object> map) {
		//use service
		Iterable<Employee> iterableEmp =
employeeMgmtService.getAllEmployees();
		//Add to model attribute
		map.put("empList", iterableEmp);
		//return LVN
		if (type.equalsIgnoreCase("pdf"))
			return "pdf_report";
		else
			return "excel_report";
	}

}
```

BootMvcProj18BeanNameViewResolverPdfExcelViewApplication.java

```
@SpringBootApplication
public class
BootMvcProj18BeanNameViewResolverPdfExcelViewApplication {

	@Bean
```

```
	@Bean
	public BeanNameViewResolver createBNVR() {

	System.out.println("BootMvcProj18BeanNameViewResolverPdfExcel
ViewApplication.createBNVR()");
		BeanNameViewResolver resolver = new
BeanNameViewResolver();
		resolver.setOrder(Ordered.HIGHEST_PRECEDENCE);
		//resolver.setOrder(1);
		return resolver;
	}

	public static void main(String[] args) {

	SpringApplication.run(BootMvcProj18BeanNameViewResolverPdfExc
elViewApplication.class, args);
	}

}
```

home.jsp

```
<%@ page isELIgnored="false" %>

<h1 style="text-align: center;">
	<a href="report?type=pdf">Employee Report as PDF</a>
</h1>

<h1 style="text-align: center;">
	<a href="report?type=excel">Employee Report as Excel</a>
</h1>
```

EmployeePDFView.java

```
package com.sahu.view;

import java.util.ArrayList;
import java.util.Map;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

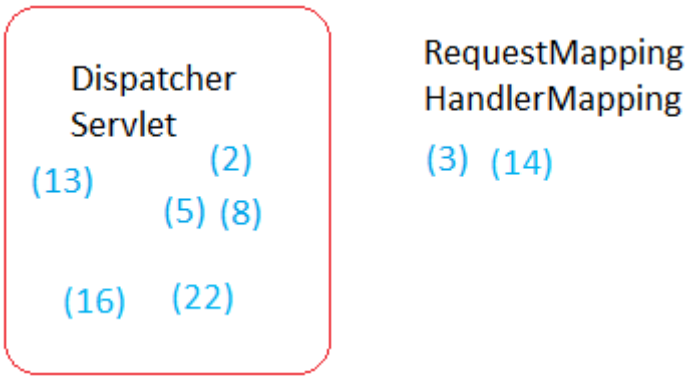
```
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.view.document.AbstractPdfView;

import com.lowagie.text.Document;
import com.lowagie.text.Font;
import com.lowagie.text.Paragraph;
import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;
import com.sahu.model.Employee;

@Component("pdf_report")
public class EmployeePDFView extends AbstractPdfView {

	@Override
	protected void buildPdfDocument(Map<String, Object> model,
Document document, PdfWriter writer,
			HttpServletRequest request, HttpServletResponse
response) throws Exception {
		//Get model attribute  data
		Iterable<Employee> empList = (Iterable<Employee>)
model.get("empList");
		//Add Paragraph
		Paragraph paragraph = new Paragraph("Employee Report",
new Font(Font.BOLDITALIC));
		document.add(paragraph);
		table content
		Table table = new Table(5, ((ArrayList<Employee>)
empList).size());
		for (Employee emp: empList) {
			//Add cells
			table.addCell(String.valueOf(emp.getEmpno()));
			table.addCell(emp.getEname());
			table.addCell(emp.getJob());
			table.addCell(String.valueOf(emp.getSal()));
			table.addCell(String.valueOf(emp.getDeptno()));
		}
		document.add(table);
	}

}
```

EmployeeExcelView.java

```
package com.sahu.view;

import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.poi.ss.usermodel.Row;
import org.apache.poi.ss.usermodel.Sheet;
import org.apache.poi.ss.usermodel.Workbook;
import org.springframework.stereotype.Component;
import org.springframework.web.servlet.view.document.AbstractXlsView;
import com.sahu.model.Employee;

@Component("excel_report")
public class EmployeeExcelView extends AbstractXlsView {

	private static int i=0;

	@Override
	protected void buildExcelDocument(Map<String, Object> model,
Workbook workbook, HttpServletRequest request,
			HttpServletResponse response) throws Exception {
		//Get model attribute
		Iterable<Employee> empList = (Iterable<Employee>)
model.get("empList");
		//Create a Sheet
		Sheet sheet1 = workbook.createSheet("Sheet1");
		//Add cells to sheet
		empList.forEach(emp->{
			//add row to sheet representing Employee record
			Row row = sheet1.createRow(i);
			row.createCell(0).setCellValue(emp.getEmpno());
			row.createCell(1).setCellValue(emp.getEname());
			row.createCell(2).setCellValue(emp.getJob());
			row.createCell(3).setCellValue(emp.getSal());
			row.createCell(4).setCellValue(emp.getDeptno());
			i++;
		});
	}

}
```

Flow of Execution:

Request URL: *http://localhost:2525/BootMVCProj18-BeanNameViewResolver-PDF-ExcelView/* (1)



EmployeeReportController.java

```
@Controller
public class EmployeeReportController {

	@Autowired
	private IEmployeeMgmtService employeeMgmtService;

	@GetMapping("/")
(6)	public String showHome() {	(4?)
		return "home";	(7)
	}

	@GetMapping("/report")
(17)	public String generateReprot(@RequestParam("type") String type,
Map<String, Object> map) {	(15?)
		//use service
(20)		Iterable<Employee> iterableEmp =
employeeMgmtService.getAllEmployees();	(18)
		//Add to model attribute
		map.put("empList", iterableEmp);
		//return LVN
		if (type.equalsIgnoreCase("pdf"))
			return "pdf_report";	(21)
		else
			return "excel_report";
	}

}
```

BootMvcProj18BeanNameViewResolverPdfExcelViewApplication.java

```
@SpringBootApplication
public class BootMvcProj18BeanNameViewResolverPdfExcelViewApplication {

	@Bean
	public BeanNameViewResolver createBNVR() {
	System.out.println("BootMvcProj18BeanNameViewResolverPdfExcelVie
wApplication.createBNVR()");
(9) (23)	BeanNameViewResolver resolver = new
(not found)                                    BeanNameViewResolver();
		resolver.setOrder(Ordered.HIGHEST_PRECEDENCE);
		//resolver.setOrder(1);
		return resolver;
	}

}
```

application.properties

```
#ViewResolver	(10) gives view object having /WEB-INF/home.jsp as the
spring.mvc.view.prefix=/WEB-INF/pages/	physical view component.
spring.mvc.view.suffix=.jsp
```

home.jsp (11)

```
<%@ page isELIgnored="false" %>
<h1 style="text-align: center;">
	<a href="report?type=pdf">Employee Report as PDF</a>
</h1>		(12)
<h1 style="text-align: center;">
	<a href="report?type=excel">Employee Report as Excel</a>
</h1>
```

EmployeeMgmtServiceImpl.java

```
@Service("empService")
public class EmployeeMgmtServiceImpl implements IEmployeeMgmtService {

	@Autowired
	private IEmployeeRepo employeeRepo;

	@Override
	public Iterable<Employee> getAllEmployees() {	(19)
```

```
		return employeeRepo.findAll();
	}

}
```

EmployeePDFView.java

```
@Component("pdf_report")     (24)
public class EmployeePDFView extends AbstractPdfView {
			calls render (-) method (25)
	@Override
	protected void buildPdfDocument(Map<String, Object> model,
Document document, PdfWriter writer,
			HttpServletRequest request, HttpServletResponse
response) throws Exception {    (26)
		//Get model attribute  data
		Iterable<Employee> empList = (Iterable<Employee>)
model.get("empList");
		//Add Paragraph
		Paragraph paragraph = new Paragraph("Employee Report", new
Font(Font.BOLDITALIC));
		document.add(paragraph);
		table content
		Table table = new Table(5, ((ArrayList<Employee>)
empList).size());
		for (Employee emp: empList) {
			//Add cells
			table.addCell(String.valueOf(emp.getEmpno()));
			table.addCell(emp.getEname());
			table.addCell(emp.getJob());
			table.addCell(String.valueOf(emp.getSal()));
			table.addCell(String.valueOf(emp.getDeptno()));
		}
		document.add(table);
	}

}
```

(27) -> render of super class (AbstractPdfView), sends response to browser and makes browser to display it as PDF document by taking MIME type as application/pdf.

Spring Boot MVC with Tiles Framework

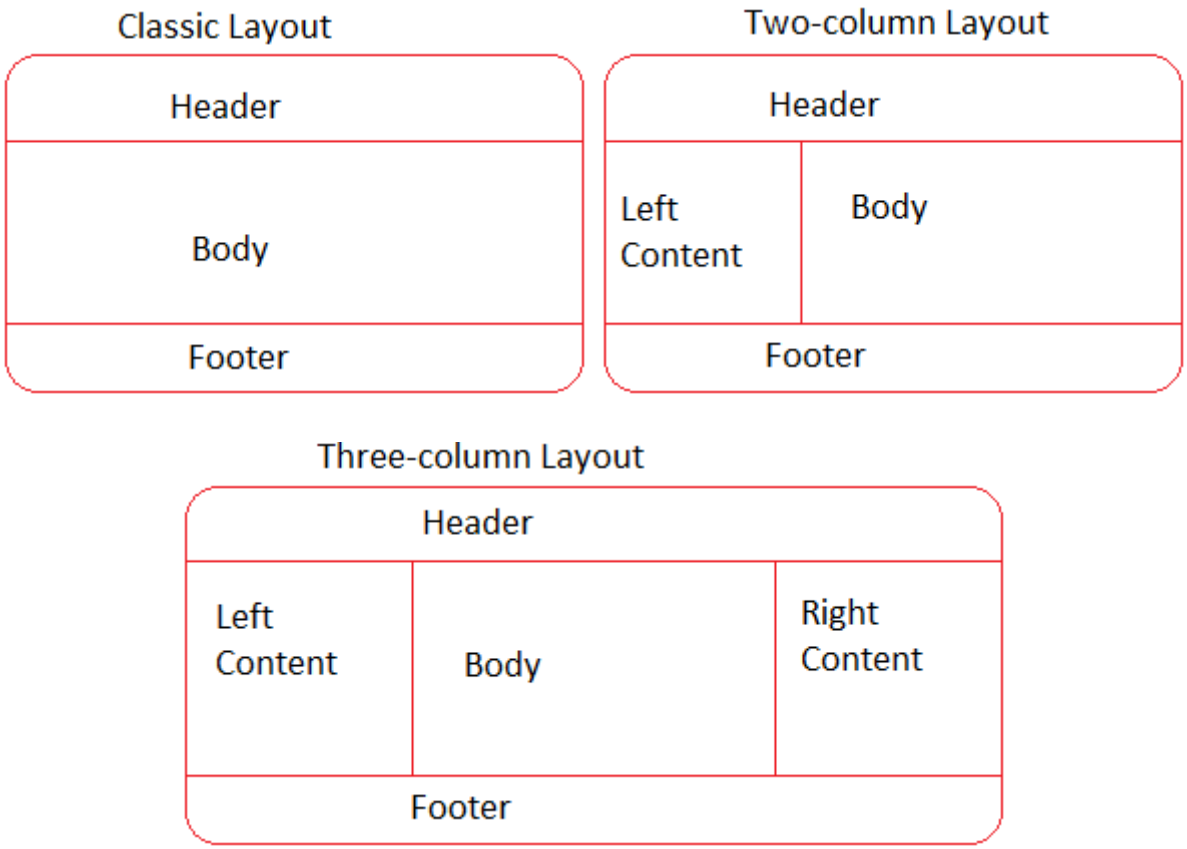
- Tiles framework is pluggable framework to any another MVC framework to display web pages of the web application having composite view Design pattern and layout control based on template page.

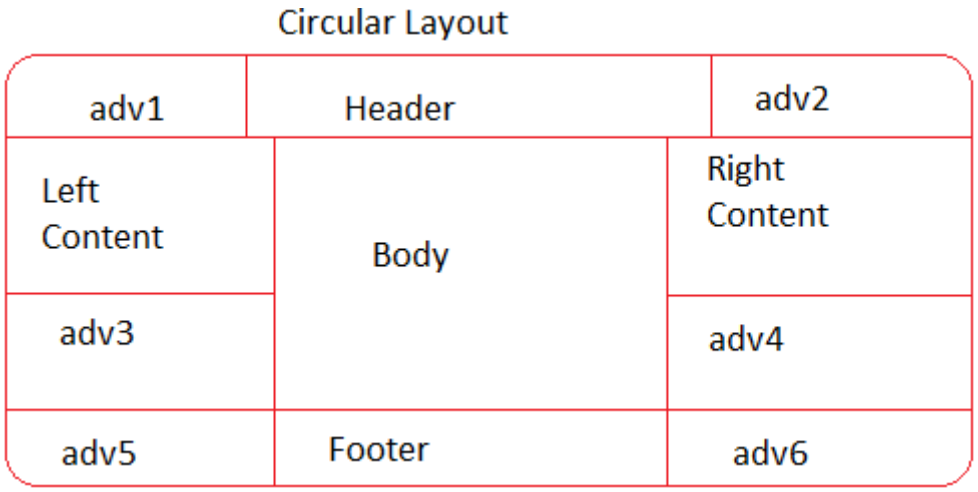
Layout controller

- All web pages of website will be created based on the common layout page/ template page to have uniform look for all the web pages. So, any change in the template page reflects to all the web pages of the website.
- Tile is logical portion of a web page.
  E.g., Header tile, Footer tile, Body tile, Left content file, Right content tile and etc.

Popular layouts in template page are

a. Classic Layout
b. Two-column Layout
c. Three-column layout
d. Circular Layout

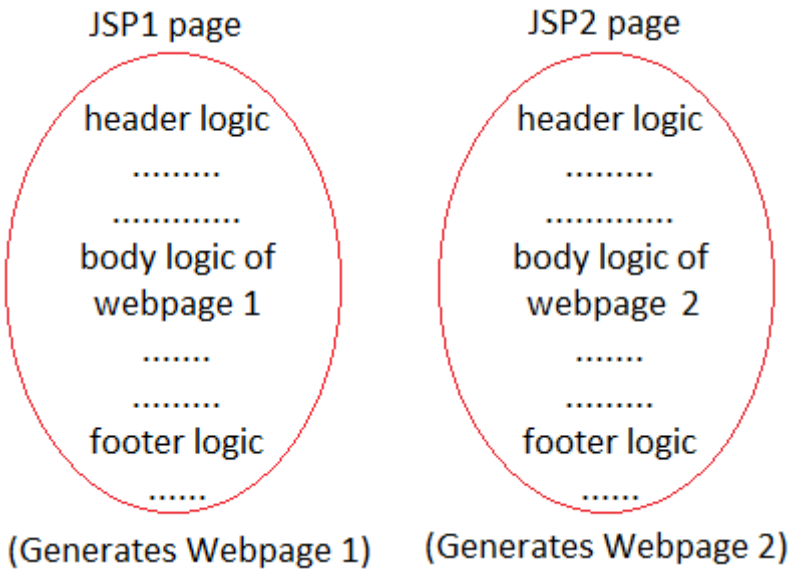




- While working with tiles framework no web page comes as the output given by single web component.
- Every web page comes having multiple tiles based on layout page containing composite output/ content given by multiple web components together.

Composite view pattern

Problem:

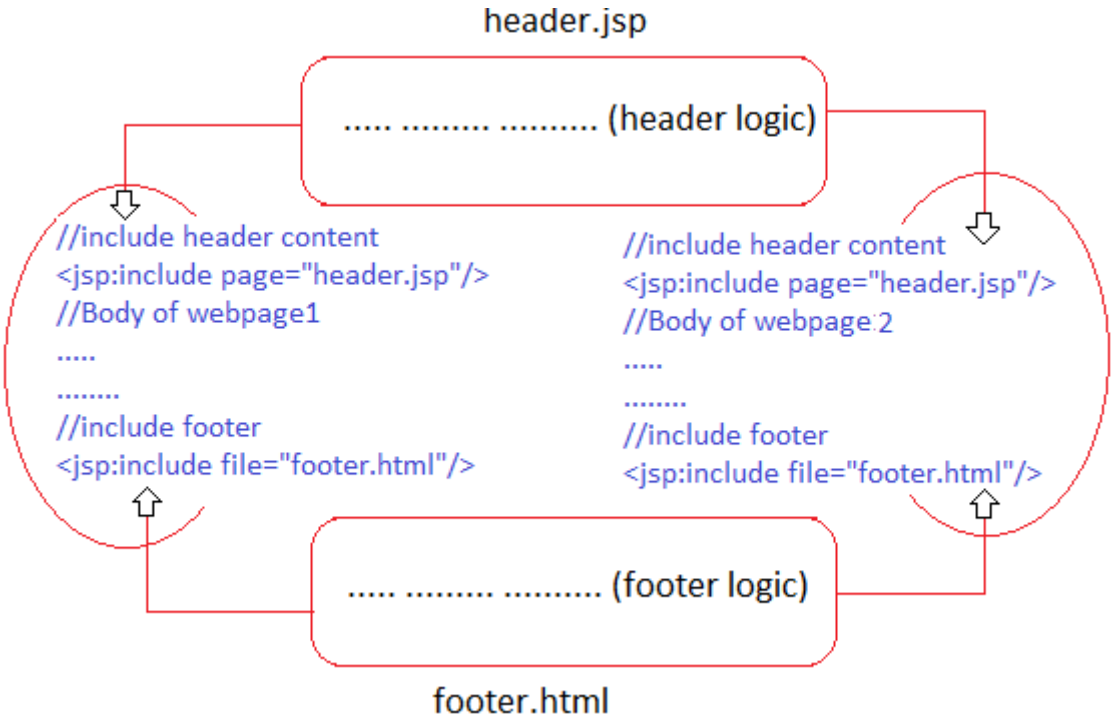


Note: Header, footer logics are not reusable.

Solution: Composite view Design pattern

- Keep common logics in separate web components and include their outputs in main web comps.

Note: header and footer logics are reusable logics.



Jsp1 -> Webpage1 (Single view)
Jsp1, Jsp2, Jsp3 pages -> webpage1 (Composite view)

Note: Tiles framework can be integrated with Struts, JSF, Spring MVC, Spring Boot MVC and etc. framework to display web pages having layout control and using composite view pattern.

Procedure to work with Tiles Framework

Step 1: Activates tiles framework in spring MVC/ Spring Boot MVC application by configurating "TilesConfigurer" as the Spring bean.
E.g.,

```
@Bean
public TilesConfigurer createTilesConfigurer() {
        TilesConfigurer configure = new TilesConfigurer ();
         configure.setDefiniations("/WEB-INF/tiles.xml");
        return configure;
}
```

Note:

- ✓ Default name and location of the tile configuration is /WEB-INF/tiles.xml file.
- ✓ WEB-INF/tiles.xml is the input file for tiles framework providing info about tile definitions.
- ✓ The TilesConfigurer simply configures a TileContainer using a set of files

containing definitions, to be accessed by TilesView instances.

Step 2: Design layout page by using “tiles” JSP tag library having “tiles” according to the chosen layout.

/WEB-INF/pages/layout.jsp (classic layout)

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%>
<table border= “0” width= “100%” height= “100%” rows= “3” cols= “1”
                                                    bgcolor= “pink”>
    <tr height= “30%” bgcolor= “grey”>
        <tiles:insertAttribute name="header"/>
    </tr>
    <tr height= “60%” bgcolor= “cyan”>
        <tiles:insertAttribute name="body"/>
    </tr>
    <tr height= “10%” bgcolor= “yellow”>
        <tiles:insertAttribute name="footer"/>
    </tr>
</table>
```

(i) displays layout.jsp having the following tiles values

/WEB-INF/pages

header.jsp

home.jsp

footer.jsp

Step 3: Decide number of pages that you want to construct based on layout page in the total website (Let’s assume 3 pages).

- Home page
- Sports page
- Entertainment page

Step 4: Develop tiles.xml having tiles definitions on 1 per each web page specifying the tile values.

tiles.xml (WEB-INF/tiles.xml)

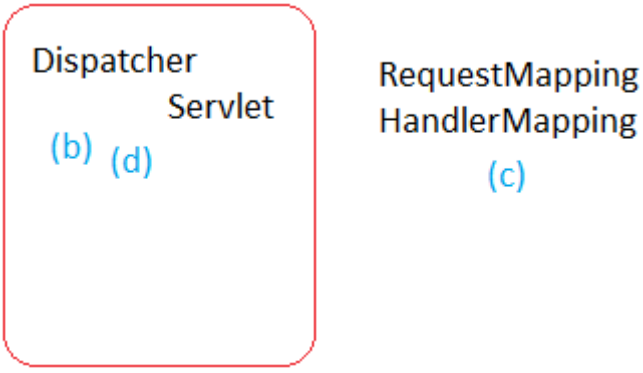
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
                        "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">
<tiles-definitions>
    <definition name="homePageDefinition" template="/WEB-
    INF/pages/layout.jsp">
        <put-attribute name="header" value="/WEB-
    INF/pages/header.jsp" />
```

```
        <put-attribute name="body" value="/WEB-
    INF/pages/home.jsp" />
        <put-attribute name="footer" value="/WEB-
    INF/pages/footer.jsp" />
    </definition>

    <definition name="sportsPageDefinition" template="/WEB-
    INF/pages/layout.jsp">
        <put-attribute name="header" value="/WEB-
    INF/pages/header.jsp" />
        <put-attribute name="body" value="/WEB-
    INF/pages/sport.jsp" />
        <put-attribute name="footer" value="/WEB-
    INF/pages/footer.jsp" />
    </definition>

    <definition name="entertainmentPageDefinition" template="/WEB-
    INF/pages/layout.jsp">
        <put-attribute name="header" value="/WEB-
    INF/pages/header.jsp" />
        <put-attribute name="body" value="/WEB-
    INF/pages/entertainment.jsp" />
        <put-attribute name="footer" value="/WEB-
    INF/pages/footer.jsp" />
    </definition>
</tiles-definitions>
```

tiles.xml (Improved using tiles inheritance)

```
<tiles-definitions>
    <definition name="baseDefinition" template="/WEB-
    INF/pages/layout.jsp">
        <put-attribute name="header" value="/WEB-
    INF/pages/header.jsp" />
        <put-attribute name="body" value="" />
        <put-attribute name="footer" value="/WEB-
    INF/pages/footer.jsp" />
    </definition>
```

(h)
```
<definition name="homePageDefinition" extends="baseDefinition">
        <put-attribute name="body" value="/WEB-
INF/pages/home.jsp" />
</definition>

<definition name="sportsPageDefinition" extends="baseDefinition">
        <put-attribute name="body" value="/WEB-
INF/pages/sport.jsp" />
</definition>

<definition name="entertainmentPageDefinition" extends="baseDefiniti
on">
        <put-attribute name="body" value="/WEB-
INF/pages/entertainment.jsp" />
</definition>
</tiles-definitions>
```



Step 5: Develop controller class having handler methods returning tile definition name as the LVN.

```
@Controller
public class showPageController {

        @GetMapping("/")
(e)     public String showHomePage(){ (c?)
                return "homePageDefinition";   (f)
        }

        @GetMapping("/sports")
```

```
        public String showSportsPage(){
                return "sportsPageDefinition";
        }

        @GetMapping("/entertainment")
        public String showEntertainmentPage(){
                return "entertainmentPageDefinition";
        }

}
```

Step 6: Configure TilesViewResolver as the Spring bean.

In Main class

```
@Bean
public TilesViewResolver createTilesViewResolver() {
        TilesViewResolver resolver = new TilesViewResolver ();
        return resolver;          (g)
}
```

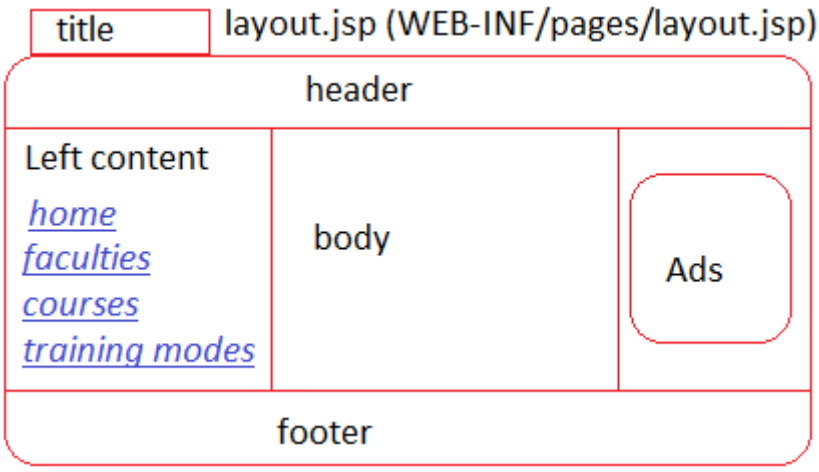
Step 7: Add the following dependencies in pom.xml file, specially related to tile implementation.

- Tiles core
- Tiles JSP

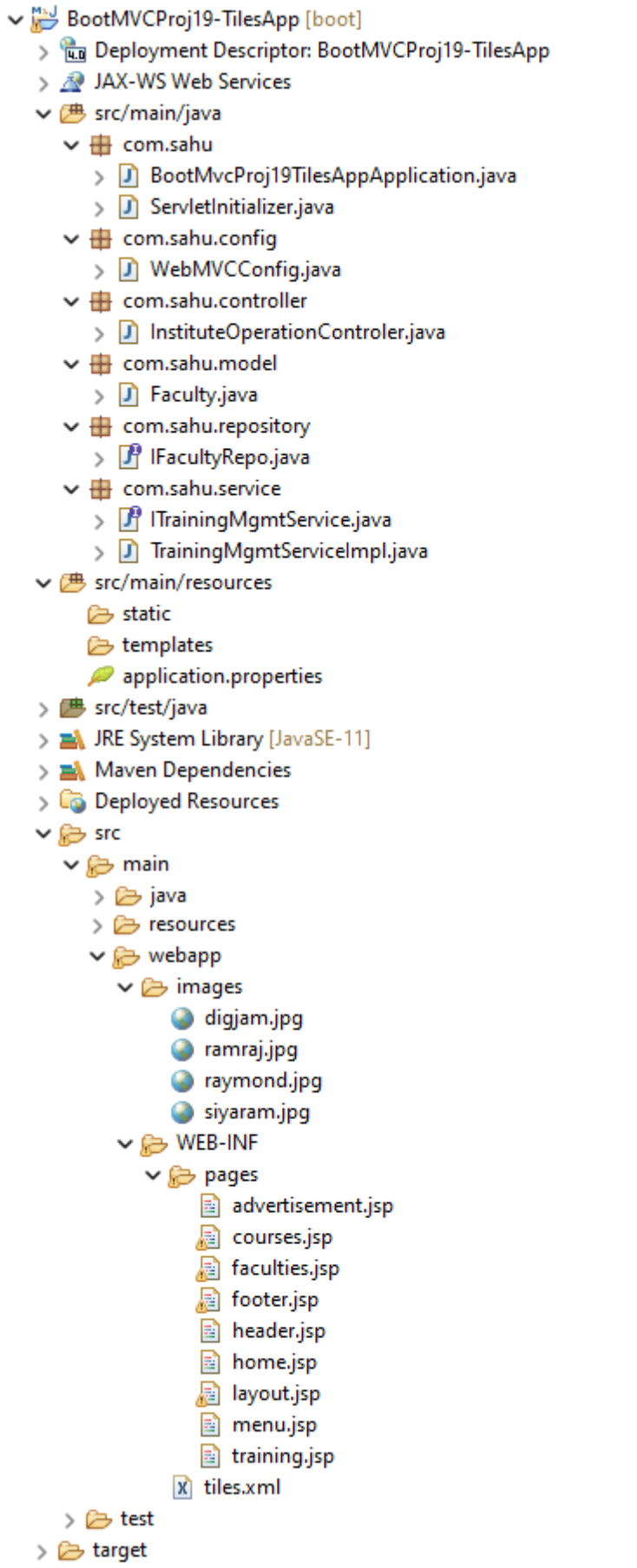
Request URL: http://localhost:2525/BootMVCProj19-TilesApp (a)

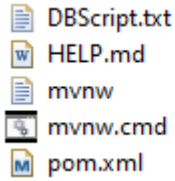
Example application:

Three-column Layout with 6 tiles

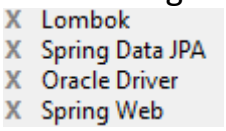


Directory Structure of BootMVCProj19-TilesApp:





- Develop the above directory structure using Spring Starter Project option and create the package, classes, folders and JSP files also
- Use following starters during project creation.



- Add the Tomcat embedded jasper jar (for embedded tomcat), Tiles core, Tile JSP, JSTL jar dependency from MVN repository in pom.xml.
- Collect the application.properties from previous project.
- Collect the images from internet.
- Then place the following code with in their respective files.

DBScript.jsp

```
CREATE TABLE "SYSTEM"."INFO_FACULTY"
(       "FID" NUMBER NOT NULL ENABLE,
        "FNAME" VARCHAR2(20 BYTE),
        "QLFY" VARCHAR2(20 BYTE),
        "SUBJECT" VARCHAR2(20 BYTE),
         CONSTRAINT "INFO_FACULTY_PK" PRIMARY KEY ("FID"));

-- Fill some content/ data
```

tiles.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE tiles-definitions PUBLIC
"-//Apache Software Foundation//DTD Tiles Configuration 2.0//EN"
                        "http://tiles.apache.org/dtds/tiles-config_2_0.dtd">

<tiles-definitions>
        <!-- Base Def -->
        <definition name="baseDef" template="/WEB-INF/pages/layout.jsp">
                <put-attribute name="title" type="string" value=""/>
                <put-attribute name="header" value="/WEB-
INF/pages/header.jsp"/>
                <put-attribute name="menu" value="/WEB-
```

```
INF/pages/menu.jsp"/>
            <put-attribute name="body" value=""/>
            <put-attribute name="advertisement" value="/WEB-
INF/pages/advertisement.jsp"/>
            <put-attribute name="footer" value="/WEB-
INF/pages/footer.jsp"/>
        </definition>

        <!-- Home Def -->
        <definition name="homePageDef" extends="baseDef">
            <put-attribute name="title" type="string" value="Home
Page"/>
            <put-attribute name="body" value="/WEB-
INF/pages/home.jsp"/>
        </definition>

        <!-- Faculties Def -->
        <definition name="facultiesPageDef" extends="baseDef">
            <put-attribute name="title" type="string" value="Faculties
Page"/>
            <put-attribute name="body" value="/WEB-
INF/pages/faculties.jsp"/>
        </definition>

        <!-- Courses Def -->
        <definition name="coursesPageDef" extends="baseDef">
            <put-attribute name="title" type="string" value="Courses
Page"/>
            <put-attribute name="body" value="/WEB-
INF/pages/courses.jsp"/>
        </definition>

        <!-- Training Def -->
        <definition name="trainingPageDef" extends="baseDef">
            <put-attribute name="title" type="string" value="Training
Page"/>
            <put-attribute name="body" value="/WEB-
INF/pages/training.jsp"/>
        </definition>

</tiles-definitions>
```

layout.jsp

```
<%@ taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles" %>

<head>
	<title>
		<tiles:insertAttribute name="title" ignore="true"/>
	</title>
</head>
<table border="0" height="100%" width="100%" rows="3" cols="3">
	<tr height="30%" bgcolor="pink">
		<td colspan="3">
			<tiles:insertAttribute name="header"/>
		</td>
	</tr>
	<tr height="60%">
		<td width="20%" bgcolor="cyan" >
			<tiles:insertAttribute name="menu"/>
		</td>
		<td width="60%" bgcolor="yellow">
			<tiles:insertAttribute name="body"/>
		</td>
		<td width="20%">
			<tiles:insertAttribute name="advertisement"/>
		</td>
	</tr>
	<tr height="10%">
		<td colspan="3" bgcolor="gray">
			<tiles:insertAttribute name="footer"/>
		</td>
	</tr>
</table>
```

footer.jsp

```
<%@ page isELIgnored="false"%>

<b>
	<i>
		<center>&copy;all rights reserverd WebTech</center>
	</i>
</b>
```

header.jsp

```
<%@ page isELIgnored="false" %>

<h1 style="color: red;  text-align: center;"> Welcome to My Web application</h1>
```

menu.jsp

```
<%@ page isELIgnored="false" %>

<h2 id="menu_link">
	<a href="./">Home</a>
</h2>
<h2 id="menu_link">
	<a href="list_faculties">All Faculties</a>
</h2>
<h2 id="menu_link">
	<a href="list_courses">All Courses</a>
</h2>
<h2 id="menu_link">
	<a href="list_training">All Training modes</a>
</h2>

<style>
#menu_link {
	text-align: left;
	margin-left: 50px;
}
</style>
```

training.jsp

```
<%@ page isELIgnored="false" %>
<h1>The Training Modes are</h1>
<ul>
	<li>Online Training</li>
	<li>Class Room Training</li>
	<li>Weekend Training</li>
	<li>1:1 Training</li>
	<li>Corporate Training</li>
</ul>
```

home.jsp

```
<%@ page isELIgnored="false" %>

<h1 style="color: red; text-align: center;">Welcome To WebTech</h1>
```

faculties.jsp

```
<%@ page isELIgnored="false" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<c:choose>
	<c:when test="${!empty facultiesList}">
		<table border="1" align="center">
			<tr>
				<th>Faculty ID</th>
				<th>Faculty Name</th>
				<th>Qualification</th>
				<th>Subject</th>
			</tr>
			<c:forEach var="fac" items="${facultiesList}">
				<tr>
					<td>${fac.fid}</td>
					<td>${fac.fname}</td>
					<td>${fac.qlfy}</td>
					<td>${fac.subject}</td>
				</tr>
			</c:forEach>
		</table>
	</c:when>
	<c:otherwise>
		<h1 style="color: red; text-align: center;">No records
found</h1>
	</c:otherwise>
</c:choose>
```

courses.jsp

```
<%@ page isELIgnored="false" %>

<h1>The Course are</h1>
<ul>
```

```
        <li>Core Java</li>
        <li>Advance Java</li>
        <li>Spring</li>
        <li>Hibernate</li>
        <li>Spring Boot & Microservice</li>
</ul>
```

advertisement.jsp

```
<%@page import="java.util.*"%>
<%@ page isELIgnored="false" %>

<%
        String adsImages[] = new
String[]{"raymond.jpg","digjam.jpg","siyaram.jpg", "ramraj.jpg"};
        int adsNumber = new Random().nextInt(adsImages.length);
        //To refresh automatic after 2 seconds
   response.setIntHeader("Refresh", 2);
%>
<img src="images/<%=adsImages[adsNumber]%>" width="100%">
```

WebMVCConfig.java

```
package com.sahu.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.view.tiles3.TilesConfigurer;
import org.springframework.web.servlet.view.tiles3.TilesViewResolver;

@Configuration
public class WebMVCConfig {

        @Bean
        public TilesConfigurer createTilesConfigurer() {
                TilesConfigurer configurer =  new TilesConfigurer();
                configurer.setDefinitions("/WEB-INF/tiles.xml");
                return configurer;
        }

        @Bean
```

```
	public TilesViewResolver createTilesViewResolver() {
		TilesViewResolver resolver = new TilesViewResolver();
		return resolver;
	}

}
```

Faculty.java

```
package com.sahu.model;

import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.Table;

import lombok.Data;

@Data
@Entity
@Table(name="INFO_FACULTY")
public class Faculty {
	@Id
	@GeneratedValue
	private Integer fid;
	private String fname;
	private String qlfy;
	private String subject;
}
```

IFacultyRepo.java

```
package com.sahu.repository;

import org.springframework.data.repository.PagingAndSortingRepository;

import com.sahu.model.Faculty;

public interface IFacultyRepo extends PagingAndSortingRepository<Faculty,
Integer> {
}
```

ITrainingMgmtService.java

```
package com.sahu.service;

import com.sahu.model.Faculty;

public interface ITrainingMgmtService {
	public Iterable<Faculty> getAllFaculties();
}
```

TrainingMgmtServiceImpl.java

```
package com.sahu.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.sahu.model.Faculty;
import com.sahu.repository.IFacultyRepo;

@Service("trainingService")
public class TrainingMgmtServiceImpl implements ITrainingMgmtService {

	@Autowired
	private IFacultyRepo facultyRepo;

	@Override
	public Iterable<Faculty> getAllFaculties() {
		return facultyRepo.findAll();
	}

}
```

InstituteOperationController.java

```
package com.sahu.controller;

import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
```

```java
import com.sahu.model.Faculty;
import com.sahu.service.ITrainingMgmtService;

@Controller
public class InstituteOperationController {

	@Autowired
	ITrainingMgmtService mgmtService;

	@GetMapping("/")
	public String showHome() {
		return "homePageDef";
	}

	@GetMapping("/list_faculties")
	public String showFacultiesPage(Map<String, Object> map) {
		//Use service
		Iterable<Faculty> itList = mgmtService.getAllFaculties();
		//Model attribute
		map.put("facultiesList", itList);
		//return LVN
		return "facultiesPageDef";
	}

	@GetMapping("/list_courses")
	public String showCoursesPage() {
		return "coursesPageDef";
	}

	@GetMapping("/list_training")
	public String showTrainingPage() {
		return "trainingPageDef";
	}

}
```

Thymeleaf

- Another UI technology that is built on the top of HTML for dynamic web pages.
- Only HTML gives static web pages. HTML tags+ thymeleaf tags gives

dynamic web pages. It is lightweight alternate for heavyweight JSP pages.

- In the execution of JSP page lot of memory and lot of CPU time is required because internally translates a JSP to equivalent Servlet component and creates multiple implicit objects (9) if used or not used for all these things lot of memory and CPU time required.
- To overcome the above problems of JSP pages use thymeleaf as lightweight alternate for rendering dynamic webpages.
- We need to write thymeleaf tags in html tags by importing thymeleaf namespace by specifying its namespace uri.

```
<html xmlns:th="https://www.thymeleaf.org">
        ……..
</html>
```

- Thymeleaf can be used only in Java environment [Thymeleaf doc].
- Thymeleaf file extension must be .html file
- Spring Boot gives built-in support of thymeleaf UI by giving
  - Default prefix is <classpath>/templates/ (Classpath here is src/main/resources folder)
  - Default suffix is .html

Note: We can't mix-up thymeleaf and JSP UI together in a single Spring MVC or Spring Boot MVC web application.

- To use thymeleaf in Spring Boot add spring-boot-starter-thymeleaf starter to pom.xml
- To execute thymeleaf code Thymeleaf engine is required which will because this jar file. This engine converts thymeleaf tags to HTML code and sends to browser as response.
- Standard JSP tags identified with fixed prefix called <jsp:xxxx> and similarly thymeleaf tags are identified with <th:xxx> prefix.
- Popular symbols in thymeleaf programming
  - @ - To specify location (/<globalpath>/<request path>)
  - $ - To read data from container managed scopes like model attributes
  - * - To bind/ link data (Model properties, reference data) to form components (useful only in thymeleaf forms)

- To read data from model attributes/ container managed scopes
  - For primitive/ wrapper/ String type model attributes th:text="${<attribute name>}"

E.g.,

- For Object type model attributes th:text="${<Object name>}"
  E.g.,
- For getting property values from Object type model attributes
  th:text="${<Object name>.<property name>}"
  E.g.,
- For looping through arrays/ collection th:each="<counter variable>:${<collection/ array type attribute>}"
- E.g., <tr th:each="emp:${empList}">

- To display images (To link image file to <img> tag)
  - <img th:src="@{/path}">

- To Link/ map with hyperlink
  - <link rel="stylesheet" th:href="@{/path}">

- To Link/ map CSS file
  - <a th:href="@{/path}"> xxxx

- To Link/ map Java script file
  - <script type="text/javascript" th:src="@{/path}"/>

Note: While working with thymeleaf, we must take controller having class level global path using @RequestMapping("/...") annotation (Global path).

- Thymeleaf forms are bidirectional forms i.e., they support Data Binding (writing form data to model class object) and Data Rendering (writing handler methods supplied model attributes/ model class object data to form components)
- To specify action URL
  - <form th:action="@{/global path/ request path}">

- For binding model class object data/ model attributes data to form components (for form backing object operation)
  - <form th:object="${model attribute name/ object name of model class}”>
    (alternate to <frm:from modelAttribute="…”>)

- For binding model class object data/ model attributes data to form components

- <input type="text" th:field="*{<model class property name/ model attribute name>}">
  (alternate to <frm:input path="…."/>

Q. What is the difference b/w <th:text> and <th:field> tags in thymeleaf?

Ans.

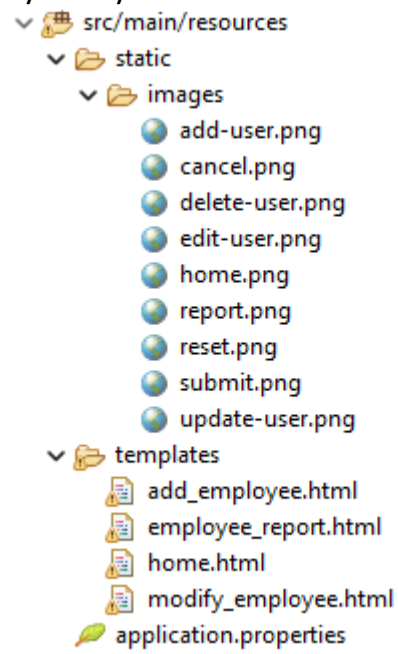
- th:text is given to read data from different scopes and to display them on browser like reading and displaying model class object data and model attributes data.
- th:field is given to bind model class object property values/ model attribute values (reference data) to form components

Converting Mini Project to Thymeleaf UI based application

Step 1: Add thymeleaf starter to pom.xml file spring-boot-starter-thymeleaf, we can remove JSTL jar files from pom.xml because we are not using JSP files.

Step 2: Provide global path to controller class.

Step 3: Copy JS, images folder of webapp or webcontent to "static" folder src/maim/resources folder and WEB-INF/pages JSP files to "template" folder of src/main/resources folder and change .jsp extension to .html.



We can comment View Resolver configuration in application.properties

Note: We can delete images, js, WEB-INF/pages folder of src/main/webapp folder.

Step 4: modify "template" folder .html files code thymeleaf code from JSP code.

Step 5: Rum the Application.

Directory Structure of BootMVCProj20-MiniProject-CURD-Thymeleaf:

- Copy and paste the BootMVCProj08-MiniProject-CURDOperation and rename to BootMVCProj20-MiniProject-CURD-Thymeleaf.
- After that change the Web Project Setting context root to BootMVCProj10-MiniProject-FormValidation.
- Then change in <artifactId> & <name> tag of pom.xml.
- Add the thymeleaf starter and you can remove the JSTL jar dependency.
- By following above procedure change file and folder structure.
- Then place the following code with in their respective files.

EmployeeController.java

```
@Controller
@RequestMapping("/employee")
public class EmployeeController {
```

home.html

```
<html xmlns:th="http://www.thymeleaf.org">

<h1 style="text-align: center;">
        <a th:href="@{/employee/emp_report}">
                <img th:src="@{/images/report.png}"/>
        </a>
</h1>
```

employee_report.html

```
<html xmlns:th="http://www.thymeleaf.org">

<div th:if="${!empsList.empty}">
        <table border="1" bgcolor="cyan" align="center">
                <tr bgcolor="pink">
                        <th>ENo</th>
                        <th>EName</th>
                        <th>Desg</th>
                        <th>Salary</th>
                        <th>Dept No</th>
                        <th>Operations</th>
                </tr>
                <tr th:each="emp:${empsList}">
                        <td><span th:text="${emp.empno}"/></td>
```

```
                    <td><span th:text="${emp.ename}"/></td>
                    <td><span th:text="${emp.job}"/></td>
                    <td><span th:text="${emp.sal}"/></td>
                    <td><span th:text="${emp.deptno}"/></td>
                    <td>
                        <a
th:href="@{/employee/edit_employee(eno=${emp.empno})}">
                            <img class="user-operation"
th:src="@{/images/edit-user.png}" />
                        </a>   
                        <a
th:href="@{/employee/delete_employee(eno=${emp.empno})}"
onclick="confirm('Do you want to delete?')">
                            <img class="user-operation"
th:src="@{/images/delete-user.png}" />
                        </a>
                    </td>
                </tr>
        </table>
</div>
<div th:if="${empsList.empty}">
        <h1 style="color: red; text-align: center;">Records not Found</h1>
</div>
<br>
<h1 class="blink_me" style="color: green; text-align: center;"
th:text="${resultMsg}"/>
<br>
<div style="text-align: center;">
        <a th:href="@{/employee/add_employee}">
            <img class="user-operation" th:src="@{/images/add-user.png}"
/>
        </a>   
        <a th:href="@{/employee/}">
            <img class="user-operation" th:src="@{/images/home.png}" />
        </a>
</div>

<style>
.user-operation {
        height: 50px;
```

```
        width: 50px;
}

.blink_me {
        animation: blinker 1s linear infinite;
}

@keyframes blinker {
        50% {
                opacity: 0;
        }
}
</style>
```

modify_employee.html

```
<html xmlns:th="http://www.thymeleaf.org">

<h1 style="color: blue; text-align: center;">Edit Employee</h1>

<form th:action="@{/employee/edit_employee}" th:object="${emp}"
method="POST">
        <table border="0" bgcolor="cyan" align="center">
                <tr>
                        <td>Employee No : </td>
                        <td><input type="text" th:field="*{empno}"
readonly="true"/></td>
                </tr>
                <tr>
                        <td>Employee Name : </td>
                        <td><input type="text" th:field="*{ename}"/> </td>
                </tr>
                <tr>
                        <td>Employee Designation : </td>
                        <td><input type="text" th:field="*{job}"/> </td>
                </tr>
                <tr>
                        <td>Employee Salary : </td>
                        <td><input type="text" th:field="*{sal}"/> </td>
                </tr>
                <tr>
```

```
                <td>Employee Dept No : </td>
                <td><input type="text" th:field="*{deptno}"/> </td>
            </tr>
            <tr>
                <td></td>
                <td>
                    <button type="reset">
                        <img class="operation-btn" alt="rest image"
th:src="@{/images/cancel.png}">
                    </button>
                      
                    <input class="operation-btn" type="image"
th:src="@{/images/update-user.png}">
                </td>
            </tr>
        </table>
</form>

<style>
.operation-btn{
        width: 50px;
        height: 50px;
}
</style>
```

add_employee.html

```
<html xmlns:th="http://www.thymeleaf.org">

<h1 style="color: blue; text-align: center;">Register Employee</h1>

<form th:action="@{/employee/add_employee}" th:object="${emp}"
method="POST">
        <table border="0" bgcolor="cyan" align="center">
            <tr>
                <td>Employee Name : </td>
                <td><input type="text" th:field="*{ename}"/> </td>
            </tr>
            <tr>
                <td>Employee Designation : </td>
                <td><input type="text" th:field="*{job}"/> </td>
```

```
            </tr>
            <tr>
                <td>Employee Salary : </td>
                <td><input type="text" th:field="*{sal}"/> </td>
            </tr>
            <tr>
                <td>Employee Dept No : </td>
                <td><input type="text" th:field="*{deptno}"/> </td>
            </tr>
            <tr> <td></td>
                <td>
                    <button type="reset">
                        <img class="operation-btn" alt="rest image"
th:src="@{/images/reset.png}">
                    </button>
                      
                    <input class="operation-btn" type="image"
th:src="@{/images/submit.png}">
                </td>
            </tr>
        </table>
</form>
<style>
.operation-btn{
        width: 50px;
        height: 50px;
}
</style>
```

Bootstrapping in Spring Boot MVC

- Bootstrapping is an UI technology that provides set of readily available CSS libraries and Javascript libraries.
- It provides multiple external CSS style classes which can be applied in our HTML/ JSP pages to improve their styles.

3 types of CSS /styles

- Inline CSS (specific to each tag)
- Embedded CSS (specific to each .html or jsp page)
- External CSS (can be used across the multiple HTML, JSP pages)

To apply Bootstrap in any JSP or HTML page

Step 1: Get Bootstrap tutorial [Official Tutorial], [w3shool tutorial].

Step 2: Import Bootstrap CSS URL with HTML or JSP page

```
<link rel="stylesheet" href="https://cdn.jsdelivr.net/npm/bootstrap@4.6.1/dist/css/bootstrap.min.css">
```

Step 3: Use Bootstrap CSS external classes in different place of HTML pages.

Directory Structure of BootMVCProj21-MiniProject-CURD-Bootstrap:

- Copy and paste the BootMVCProj20-MiniProject-CURD-Thymeleaf and rename to BootMVCProj21-MiniProject-CURD-Bootstrap.
- After that change the Web Project Setting context root to BootMVCProj21-MiniProject-CURD-Bootstrap.
- Then change in <artifactId> & <name> tag of pom.xml.
- Then place the following code with in their respective files.

employee_report.html

```
<html xmlns:th="http://www.thymeleaf.org">

<head>
	<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.1.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-1BmE4kWBq78iYhFldvKuhfTAU6auU8tT94WrHftjDbrCEXSU1oBoqyl2QvZ6jIW3" crossorigin="anonymous">
</head>

<div class="container" th:if="${!empsList.empty}">
	<table class="table table-hover" border="1" bgcolor="cyan" align="center">
		<tr class="bg-danger text-success" >
			<th>ENo</th>
			<th>EName</th>
			<th>Desg</th>
			<th>Salary</th>
			<th>Dept No</th>
			<th>Operations</th>
		</tr>
		<tr class="bg-primary text-white" th:each="emp:${empsList}">
			<td><span th:text="${emp.empno}"/></td>
			<td><span th:text="${emp.ename}"/></td>
```

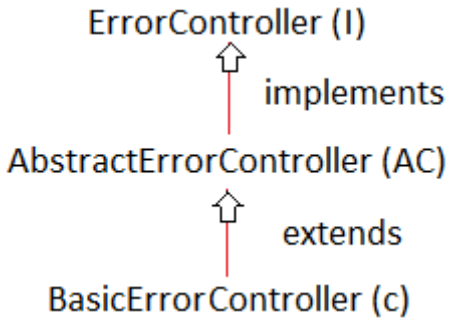
```
                    <td><span th:text="${emp.job}"/></td>
                    <td><span th:text="${emp.sal}"/></td>
                    <td><span th:text="${emp.deptno}"/></td>
                    <td>
                        <a
th:href="@{/employee/edit_employee(eno=${emp.empno})}">
                            <img class="user-operation"
th:src="@{/images/edit-user.png}" />
                        </a>   
                        <a
th:href="@{/employee/delete_employee(eno=${emp.empno})}"
onclick="confirm('Do you want to delete?')">
                            <img class="user-operation"
th:src="@{/images/delete-user.png}" />
                        </a>
                    </td>
                </tr>
        </table>
</div>
<div th:if="${empsList.empty}">
        <h1 style="color: red; text-align: center;">Records not Found</h1>
</div>
<br>
<h1 class="blink_me" style="color: green; text-align: center;"
th:text="${resultMsg}"/>
<br>
<div style="text-align: center;">
        <a th:href="@{/employee/add_employee}">
            <img class="user-operation" th:src="@{/images/add-user.png}"
/>
        </a>   
        <a th:href="@{/employee/}">
            <img class="user-operation" th:src="@{/images/home.png}" />
        </a>
</div>
<style>
.user-operation {
        height: 50px;
        width: 50px;
}
```

Error or Exception handling in Spring MVC/ Spring Boot MVC

HTTP status/ Response status codes	Details
101-199/ 1XX	Information (The happenings in the server will be displayed).
200-299/ 2XX	Success (For given request, the response has come successfully without error/ exception).
300-399/ 3XX	Redirection (Given request to one website it will be redirected to another website).
400-499/ 4XX	Incomplete (Some problem in the locating and executing web component/ Client-side errors).
500-599/ 5XX	Server Error (For exceptions raised in web components or in underlying server/ container).

Note: 400-499 or 500-599, these codes will come when there is a problem in the execution of web application and also called as HTTP errors.

- Spring Rest/ Spring Boot MVC is having pre-defined Controller to handle these HTTP Errors/ exception and to give default White Label error pages.

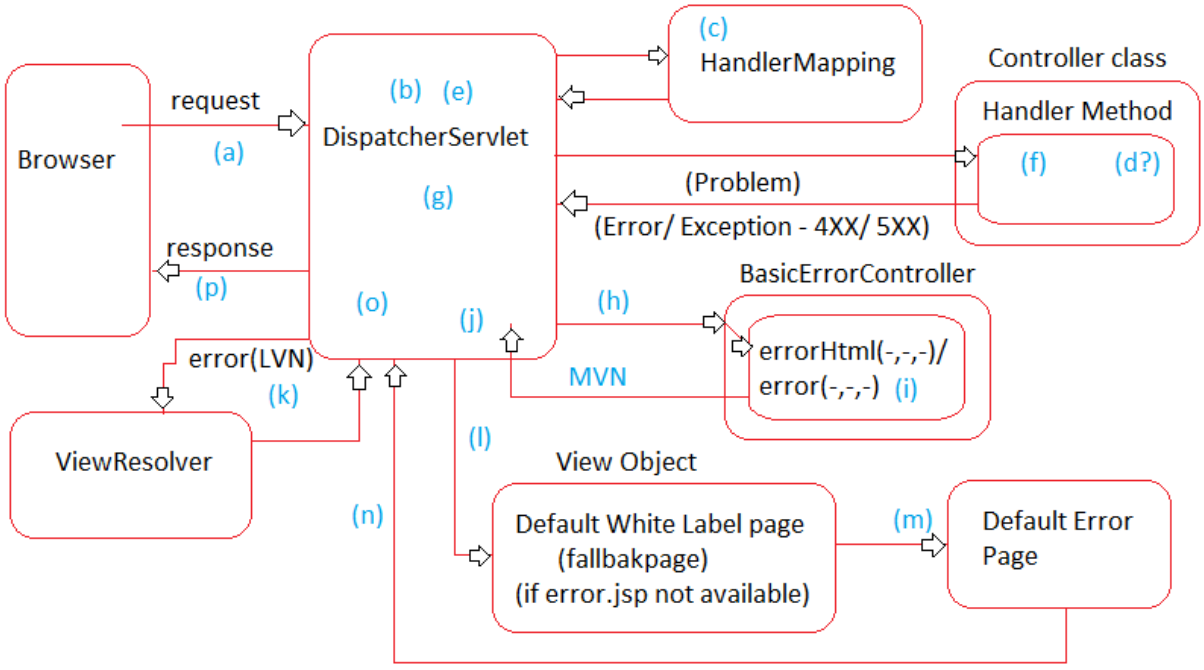


BasicErrorController

- Request path is (<contextpath>/error)
- This ErrorController can handle both Spring Boot MVC & Spring Rest application HTTP Errors/ Exceptions and generates White Label Error pages.
- This class have two important methods
  - errorHtml (-, -, -) (return type is ModelAndView) [Browser generated requests for Spring MVC/ Spring Rest application)

- error (-, -, -) (return type is ResponseEntity) [Non-browser generated requests for Spring MVC/ Spring Rest application Desktop app/ Mobile app/ Tools (Postman), etc.]

Flow of Error/ Exception handling in Spring Boot MVC



- Instead of White Label error page which is fallback error page for different problems we can configure custom Error Pages for different HTTP Errors/ Exceptions (4XX and 5XX problems)
- We configure error.jsp in WEB-INF/pages folder as global Error page i.e., these error page executes for 4XX and 5XX error/ exceptions raised in web application.
- We can configure error page for each error code like 404, 405, 500, 501, 502 and etc. for that we need to take <errorcode>.jsp as the file name in WEB-INF/pages/error folder.
- We can also take error page for specific series error codes like 4xx.jsp for 400-499 error codes and similarly 5xx.jsp for all 500-599 errors/ exceptions, we should also these pages in WEBE-INF/pages/error folder.
- We can also make Custom Exceptions generating our choice error codes (4XX or 5XX) by using @ResponseStatus annotation on the top of Custom Exception class.

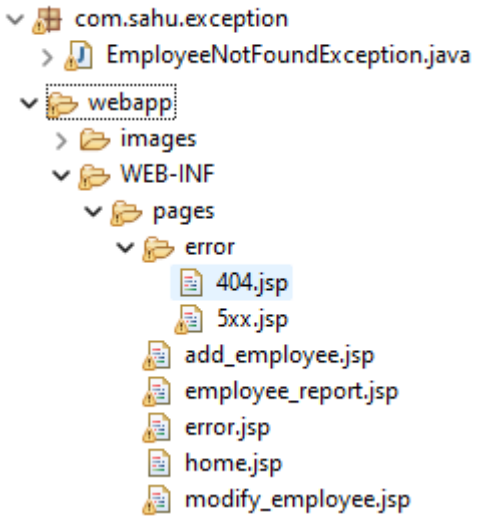
Order of executing error pages:

1. Problem in Handler method
2. Specific error code page (like 404.isp, 500.jsp, 501.jsp and etc.)

3. Specific error series page (4xx.jsp, 5xx.jsp)
4. Global error page (error.jsp)
5. Default White Label error page

Directory Structure of BootMVCProj22-MiniProject-CURD-ErrorPage:

- Copy and paste the BootMVCProj08-MiniProject-CURDOperation and rename to BootMVCProj22-MiniProject-CURD-ErrorPage.
- After that change the Web Project Setting context root to BootMVCProj22-MiniProject-CURD-ErrorPage.
- Then change in <artifactId> & <name> tag of pom.xml.
- Add the following package, class, folder, JSP file under the exact location like shown below to implement above points.



- Collect required images from internet.
- Then place the following code with in their respective files.

EmployeeNotFoundException.java

```
package com.sahu.exception;

import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(code = HttpStatus.NOT_IMPLEMENTED)
public class EmployeeNotFoundException extends RuntimeException {

	public EmployeeNotFoundException(String message) {
		super(message);
	}

}
```

EmployeeMgmtServiceImpl.java

```
@Override
	public Employee getEmployeeByEmpNo(Integer empno) {
		/*Optional<Employee> opt = employeeRepo.findById(empno);
		if (opt.isPresent())
			return opt.get();
		else
			throw new EmployeeNotFoundException("Employee
with "+empno+" not found.");*/
		//Alternet code
		return employeeRepo.findById(empno).orElseThrow(()->new
EmployeeNotFoundException("Employee with "+empno+" not found."));
	}
```

error.jsp

```
<%@ page isELIgnored="false" %>
<h1 style="color: red; text-align: center;">Some Problem</h1>

<div style="text-align: center;">
	<img alt="sorry image" src="images/sorry-page.jpg">
</div>
<table border="1" bgcolor="pink" align="center">
	<tr>
		<td>Status</td>
		<td>${status}</td>
	</tr>
	<tr>
		<td>Type</td>
		<td>${type}</td>
	</tr>
	<tr>
		<td>Time stamp</td>
		<td>${timestamp}</td>
	</tr>
	<tr>
		<td>Path</td>
		<td>${path}</td>
	</tr>
</table>
```

400.jsp

```
<%@ page isELIgnored="false" %>
<h1 style="color: red; text-align: center;">404 Error</h1>

<div style="text-align: center;">
	<img width="100%" alt="404 image" src="images/404.png">
</div>
```

5xx.jsp

```
<%@ page isELIgnored="false" %>
<h1 style="color: red; text-align: center;">5XX Error</h1>

<div style="text-align: center;">
	<img alt="sorry image" src="images/5XX.jpg">
</div>

<table border="1" bgcolor="pink" align="center">
	<tr>
		<td>Status</td>
		<td>${status}</td>
	</tr>
	<tr>
		<td>Type</td>
		<td>${type}</td>
	</tr>
	<tr>
		<td>Time stamp</td>
		<td>${timestamp}</td>
	</tr>
	<tr>
		<td>Path</td>
		<td>${path}</td>
	</tr>
	<tr>
		<td>Message</td>
		<td>${message}</td>
	</tr>
</table>
```

Servers in Spring Boot Web applications

- The Spring Boot MVC/ Spring Boot Rest applications can be deployed and executed in two types of servers
    a. Spring Boot web Embedded servers
        - Apache Tomcat (default) (popular)
        - Eclipse Jetty
        - JBoss Undertow

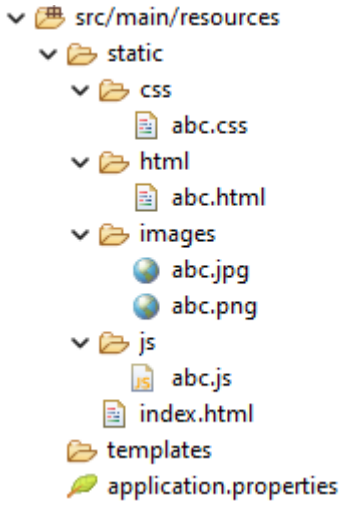
    b. External servers
        - Apache Tomcat (3)
        - Jetty
        - Wildfly (1)
        - Glassfish (2)
        - Undertow
        - WebLogic
        - WebSphere
        - JBoss

        and etc. (In fact, any Java-based web server or application server can be used)

- Spring Boot is not given to develop EJB components, so we do not need EJB container for Spring Boot applications.
- Spring Boot is given to develop standalone apps, web applications, Distributed apps (Restful-web based) which are generally packaged as jar files/ war files.
- So, to deploy and execute Spring Boot MVC apps/ Spring Rest apps there is no need of Java based application servers, we can manage with Java based web servers. For this reason, they have not given any application server as embedded server in Spring Boot.
- EAR file (Enterprise Application Archive)
  ear file = jar file (ELB component) + war file (web application)
- Spring Boot does not support ear files deployment generally we need application server to deploy and execute ear files.
- So, to deploy and execute Spring Boot MVC/ Spring Rest app we just need web servers like tomcat, jetty, undertow and etc.
- In order to work with other embedded servers in Spring Boot MVC applications we need to disable spring-boot-tomcat-starter dependency from pom.xml file and we need to add other embedded server's starter dependencies to pom.xml file.
- We can place static content in Spring Boot MVC application in "static"

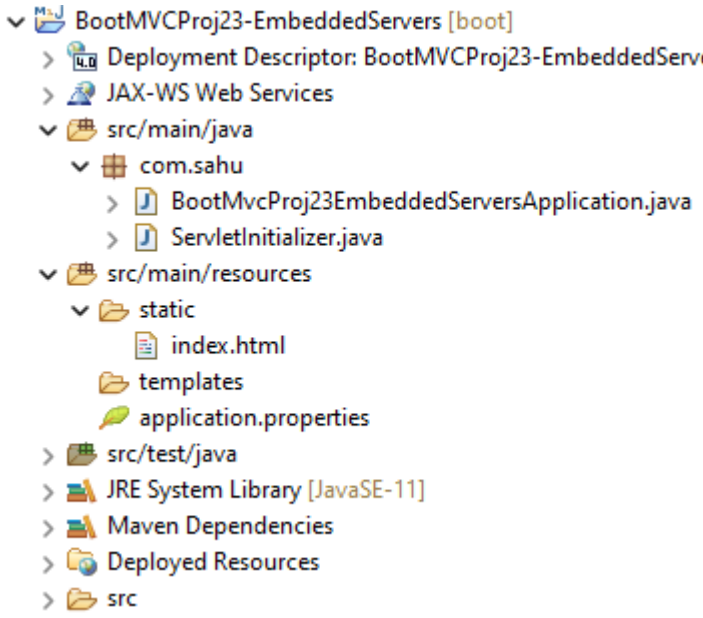
folder of src/main/resources folder.

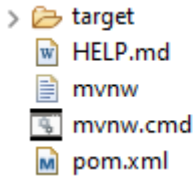
- Static content means "CSS", "JS", "images", "HTML", "audio", "video" and etc. files.
- In most of servers the index.html placed in "static" folder of "src/main/resources" folder will be taken as default welcome file.



- If we add spring-boot-web-starter by selecting "Spring web" towards creating Spring Boot project then the spring-boot-tomcat-starter dependency will come automatically, which gives embedded tomcat server.
- In recent version of Spring Boot, spring-boot-tomcat-starter is coming an independent and separate starter, earlier it used to come as dependent jar file for spring-boot-web-starter dependency.

Directory Structure of BootMVCProj23-EmbeddedServers:





- Develop the above directory structure using Spring Starter Project option and create the HTML files.
- Use Spring web starter during project creation.
- Then place the following code with in their respective files.

application.properties

```
server.port=4041
server.servlet.context-path=/serverapp
```

index.html

```
<!DOCTYPE html>
<html>
<head>
	<title>Welcome Page</title>
</head>
<body>
	<h1 style="color: blue; text-align: center;">Welcome to Spring Boot</h1>
</body>
</html>
```

Note: Run the application as Spring Boot App or Java Application, for that right click on the project then Run As, Spring Boot App
Give the URL in browser: http://localhost:4041/serverapp/

Work with Jetty server as Embedded server

Step 1: Disable spring-boot-tomcat-starter dependency from pom.xml.
Step 2: Add spring-boot-jetty-starter dependency to pom.xml [Link].

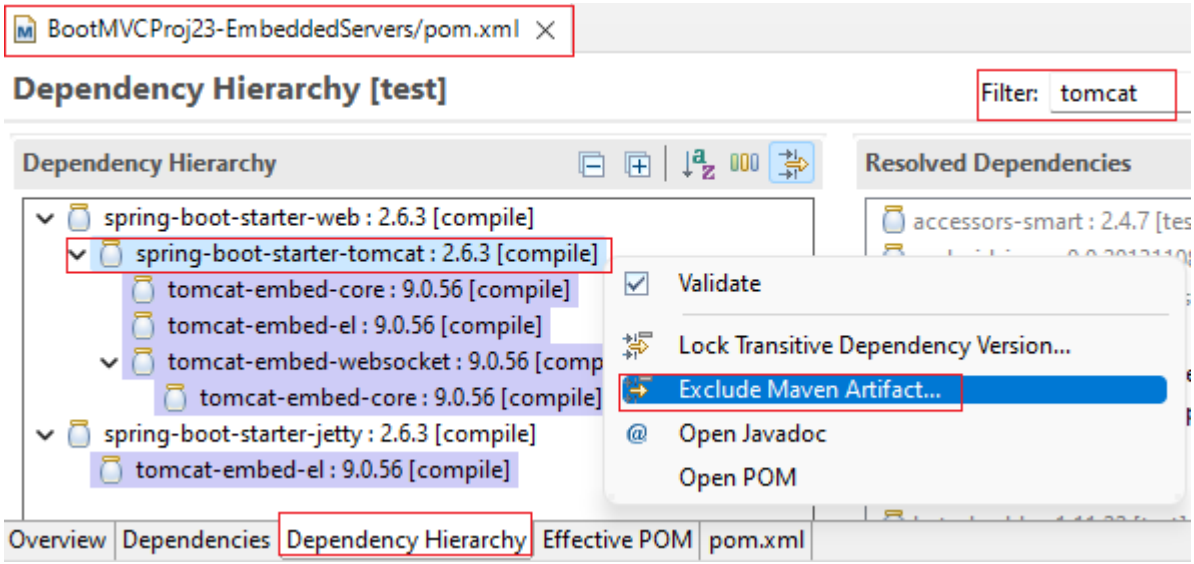
pom.xml

```
		<dependency>
			<groupId>org.springframework.boot</groupId>
			<artifactId>spring-boot-starter-jetty</artifactId>
		</dependency>
```

Note:

- ✓ Still, it will run using Embedded tomcat server because in now versions they give the spring-boot-start-tomcat outside as separate jar and as well as with spring-boot-starter-web.
- ✓ So, only commenting the outside dependency is not sufficient we have to exclude from spring-boot-starter-web also for the follow the below steps.

Step: Go to pom.xml click on Dependency Hierarchy then in Filter search for tomcat then right click on spring-boot-starter-tomcate-2.6.3, click on Exclude Maven Artifact.



Work with "JBoss Undertow" server as Embedded server

Step 1: Same as above.

Step 2: Add "spring-boot-stater-undertow" to pom.xml [Link].

pom.xml

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-undertow</artifactId>
</dependency>
```

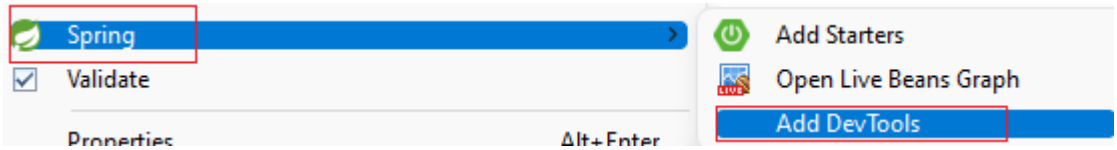
Note: If we add all the 3 embedded server's dependency starters to pom.xml file then the Spring Boot MVC app chooses the embedded server in the following order

a. Tomcat

b. Jetty (if Tomcat dependency is not available)
c. Undertow (if Tomcat, Jetty dependencies are not available)

"Dev Tools" support to Spring Boot MVC/ Spring Rest

- Generally, when we do source code changes (especially Java code) in Spring Boot web MVC, Spring Rest applications we need to restart the server to reflect those changes.
- Some IDES like eclipse automatically reloads the web application for code changes but that is not so effective some times. Which again forces us to go for clean project, restart IDE and etc. addtional activities.
- Sometimes these IDEs will not recognize the changes done static content especially CSS files, JS files.
- To overcome this problem and simplify code modifications in the development mode of the project, it is recommended to add "spring-boot-devtools" dependency to pom.xml.
- We can add this stater as the dependency while creating the project or after creating project using (right click on project then go to Spring, click on Add DevTools)



Note: The moment we try to save changes done in .java files using "save" button or CRTL+S key, the dev tools recognize the changes in .class files (byte code) and internally uses some live parallel server to reflect the changes by restarting the servers and reloading the applications.

---------- The END ----------