Control Flow of SpringBootMVC
==========
1. Programmer deploys SpringMVC/SpringBootMVC application in webserver or webapps
2. Deployment activities takes place which involves IOC container creation,DispatcherServlet registration with ServletContainer
    Pre-Instantiation of Singleton scope spring beans like Controller class,handlermapping,viewResolvers,Service class,DAO class etc.
    In the mean time necessary dependancy injection also takes place
3. Browser gives request to deployed springmvc application
4. The frontcontroller(DispatcherServlet) traps the request and applies the common system services on the request
      like logging,auditing,tracking etc,...
5. DispatcherServlet hands over the request to HandlerMapping component to map incoming request with handler method of handler/controller
    class and gets RequestMapping object from HandlerMapping component having Controller class bean id and HandlerMethod signature(it uses lot
    of reflection api code internally)
6. DispatcherServlet takes controller bean class id from the recieved RequestMapping object and gets the Controller class object from
    DispatcherServlet created IOC container.DispatcherServlet also prepares the necessary objects based on the method signature of the
      handled method collected from RequestMapping Info object.
7. DispatcherServlet calls handler method having necessary object as the arguments on the above received Controller class object.
8. The handler method of the controller class either directly process the request or takes the support of service/DAO and keeps the result
    in a scope(prefereably in request scope)
9. The handler method of Controller class returns LVN(logical view name) back to Dispatcher Servlet.
10. DispatcherSerlvet gives LVN to ViewResolver
11. ViewResolver map/link LVN to PhysicalView component and returns View Object having PhysicalViewName and location.
12. DispatcherServlet gets PhysicalViewName and location from the recieved View Object and send the control to Physical view component
     using rd.forward(,) to format the results gathered from particular scope(preferabley request scope) using presentation logics.
     These formatted results goes back to DispatcherServlet.
13. DispatcherServlet sends the formatted result to browser as the response.

Keypoints about SpringBootMVC application
==========
=> SpringBoot MVC can be developed as Standalone webapp or deployable webapplication in webserver/Application Server.
=> SpringBoot MVC Standalone web application means nothing but webapplication that uses "tomcat embeded server" so no need of arranging
   any external server explicitly[Good in development,testing environment]
=> This application comes as a jar file,here SpringApplication.run() itself creates tomcat embeded server along with IOC container creation
   and DispatcherServlet Registration, no need of sepearte ServletInitializer class.
=> SpringBootMVC deployable webapplication is nothing but webapplication that is deployable in any webserver as normal war file.
=> Here Seperate ServletInitializer class is required extending from SpringBootServlet Initializer to create IOCcontainer and register
   DispatcherServlet dynamically.
=> In both the cases IOC container created is "AnnotationConfigWebApplicationContext".

Note:: While creating Spring-boot-mvc project if we select the type as "war", then the starter files will create 2 java files

a. Main class cum Configuration class having @SpringBootApplication
=> The main() class is not required if we are running our App as Deployable webapplication using External Tomcatserver/any other server

b. ServletInitializer extending SpringBootServletInitializer
=> This class is not required if we are running our application as standalone webapplication using embeded server.

```
BootMVCProj1-DisplayingHomePage
            |=> src/main/java
                        |=> in.ineuron
                                    |=>
BootMvcProj1DisplayingHomePageApplication(@SpringBootApplication)
                                    |=> ServletInitializer(extends
SpringBootServletInitializer)
```

Note: If we run springboot application as standalone app,then the SpringApplication.run() method takes care of multiple activities like

a. Creating an Embeded tomcat server
b. Creating an IOC container
c. Creating an DispatcherServlet having IOC-container
d. Creating an ErrorFilter classes objects
e. Registering DispatcherServlet,ErrorFilters dynamically with Servlet Container
so on.....

=> By default SpringBootMVC Apps runs without ContextPath, To provide ContextPath for those Apps we need to add one special entry in application.properties

application.properties
```
server.servlet.context-path=/FirstApp
spring.mvc.view.prefix = /WEB-INF/pages
spring.mvc.view.suffix = .jsp
server.port = 9999

old url :: http://localhost:9999/index.jsp
new url :: http://localhost:9999/FirstApp/index.jsp
```

Note: Embeded tomcat server is not taking index.jsp as the default welcome file, but external tomcat server will take

Only standalone execution of springbootmvc app takes the cfg "ContextPath" from application.properites file

The deployable webapplication execution in external server takes the project name as the "ContextPath".

If we are running SpringBootMVC app as a deployable app in external server, then we can comment main() of @SpringBootApplication class

If we are running SpringBootMVC app as a standalone app using embeded server then we can remove "ServletInitalizer" class.

In order to get welcomepage/home page of the spring mvc app without typing index.jsp in the browser address(stanadlone webapp) or to avoid index.jsp,we need to take the handler method that give LVN of home page having request path="/".
eg::
```
@Controller
```

```
public class ShowHomeController {
      @RequestMapping(value = "/", method = RequestMethod.GET)
      public String showHomePage() {
            return "home"; //WEB-INF/pages/home.jsp[delete index.jsp avaialble in
public area]
      }
}
request url:: http://localhost:9999/FirstApp/
output:: Welcome to SpringMVC

Working with Handler Methods
============================
 => If the handlerMapping method return type is "String" then it returns only "LVN"
to dispatcher Servlet.
 => If the handlerMapping method return type is "ModelAndView" then it return
"Model Attributes" and "LogicalViewName" to DispatcherServlet.
 => If the DispatcherServlet receives ModelAndView Class Object from the Handler
method then it makes the model attribute
    as the request scope data and it gives the recieved LVN to ViewResolver.
 => Taking ModelAndView is a legacy approach, go for alternative.

application.properties
======================
server.port=9999
spring.mvc.view.prefix=/WEB-INF/pages/
spring.mvc.view.suffix=.jsp
server.servlet.context-path=/SecondApp


eg#1.
@RequestMapping(value="/",method = RequestMethod.GET)
public String showHome() {
            return "home";//WEB-INF/pages/home.jsp
}

Usage of ModelAndView
=====================
@RequestMapping(value = "/wish",method = RequestMethod.GET)
public ModelAndView generateMessage() {
            String msg = service.generateWishMessage();
            ModelAndView mav = new ModelAndView();
            mav.addObject("wmg", msg);
            mav.setViewName("display");// WEB-INF/pages/display.jsp
            return mav;
}

request url:: http://localhost:9999/SecondApp/

Usage of Model(I)
================
@RequestMapping(value = "/wish", method = RequestMethod.GET)
public String generateMessage(Model model) {
      System.out.println("SharedMemory object class name is :: " +
model.getClass().getName());

      String msg = service.generateWishMessage();
      model.addAttribute("wmg",msg);

      return "display"; //WEB-INF/pages/display.jsp
```

```
}

Usage of Map<String,Object>[BestApproach as it is Non-Invasive]
================================================================
@RequestMapping(value = "/wish", method = RequestMethod.GET)
public String generateMessage(Map<String,Object> model) {
      System.out.println("SharedMemory object class name is :: " +
model.getClass().getName());

      String msg = service.generateWishMessage();
      model.put("wmg",msg);

      return "display";
}

Making the return type as void
==============================
@RequestMapping(value = "/wish", method = RequestMethod.GET)
public void generateMessage(Map<String, Object> model) {
      System.out.println("SharedMemory object class name is :: " +
model.getClass().getName());

      String msg = service.generateWishMessage();
      model.put("wmg", msg);//WEB-INF/pages/wish.jsp

}
If the handler method return type is void,it takes the incoming request path as
excluding("/") as LVN by using "RequestToViewNameTranslator"
concept internally.
      eg:: incoming reuqest is "/wish", then it takes "wish" as LVN.


Behind the scenes
=================
=> DispatcherServlet creates one SharedMemory called "BindingAwareModelMap" object
to maintain model attributes having request scope.
=> By exposing this BindingAwareModelMap objects to handler methods as a parameter
we can add model data attributes to it
   (default scope is request)
=> All view components belonging to same request scope can read and use model
attributes data.


Note: If we are using InternalResourceViewResolver in SpringMVC applications, then
all the view components must be in the same location
      (prefix) and in same technology(suffix).
      To keep different view comps in different locations and in different
technologies take the support of Multiple view resolvers.

Sending the response directly to the browser without using DispatcherServlet
============================================================================
@RequestMapping(value="/wish", method = RequestMethod.GET)
 public void generateWishMessage(HttpServletResponse response) throws Excpetion{
      String msg = service.generateWishMessage();
      PrintWriter out = response.getWriter();
      response.setContentType("text/html");
      pw.println("<b>WishMessage is :: "+msg+"</b>");
      return ;
 }
```

```
@RequestMapping(value="/wish", method = RequestMethod.GET)
public String generateWishMessage(HttpServletResponse response) throws Excpetion{
      String msg = service.generateWishMessage();
      PrintWriter out = response.getWriter();
      response.setContentType("text/html");
      pw.println("<b>WishMessage is :: "+msg+"</b>");
      return null;
}
```

These type of handler methods are so useful when we perform "FileDownloading" activities.

eg::

```
 @RequestMapping(value="/wish", method = RequestMethod.GET)
 public Model generateWishMessage(){

      String msg = service.generateWishMessage();
      Model model = new ExtendedModelMap();
      model.addAttribute("wmg",msg);
      return model;
 }
```

Taking Model as the return type of HandlerMethod is a bad practise,because of the following reasons

a. We should know the process of Creation Model object explicitly
b. Internally created SharedMemory(BindingAwareModelMap) will be wasted
c. There is no control on LVN, we are forced to take the request uri as the logical name.

Taking Model as the parameter of the HandlerMethod is a good practise,because of the following reasons

a. Full Control of LVN.
b. Creating BindingAwareModelMap object as the sharedmemory is taken care by DispatcherServlet
c. Creating parameter type reference variable pointing to the sharedmemory is also taken care by DispatcherServlet.
d. Internally created SharedMemory(BindingAwareModelMap) will not be wasted.
e. Taking Parameter type as HashMap or Map we make Handler methods as "Non-Invasive".

We need to consider the following points while giving requestpath to handler methods of Controller class

---

a. The request path of handler method must start with "/".
b. The request path of handler method is caseSensitive.

eg::

```
@Controller
public class TestController{

      @RequestMapping(value= "/REPORT)
      public String showReport(Map<String,Object> map){
                  ;;;;
      }

      @RequestMapping(value= "/report)
      public String showReport1(Map<String,Object> map){
                  ;;;;;
```

```
	}
}

requestURL :: http://localhost:9999/ThirdApp/REPORT ===> showReport()
requestURL :: http://localhost:9999/ThirdApp/report ===> showReport1()

c. Multiple request have same request path with different request methods like
POST/GET

@Controller
public class TestController {
	@GetMapping
	public   String  showHome() {
		System.out.println("TestController.showHome()");
		return "home";
	}

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

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

home.jsp
========
<%@ page isELIgnored="false" %>
	<!--POST REQUEST-->
<form action="report"  method="POST">
	<input type="submit" value="send"/>
</form>
<br>
<a href="report">link1</a> <!--GET REQUEST-->

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

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

Note:: Instead of using @RequestMapping by specifiying requestmode GET/POST we can
directly use "@PostMapping" and "GetMapping".

d. Taking @RequestMapping without requestpath  takes "/" as default path

@GetMapping("/")
public   String  showHome() {
	System.out.println("TestController.showHome()");
	return "home";
}
	is equal to
```

```
@GetMapping
public   String  showHome() {
      System.out.println("TestController.showHome()");
      return "home";
}
```

e. One Handler class can have multiple request paths

```
@GetMapping(value={"/report","/report1","/report2"})
public  String showReport(Map<String,Object> map) {
            System.out.println("TestController.showReport(-)");
            return "display";
}

requestURL :: http://localhost:9999/ThirdApp/report  ===> showReport()
requestURL :: http://localhost:9999/ThirdApp/report1 ===> showReport()
requestURL :: http://localhost:9999/ThirdApp/report2 ===> showReport()
```

f. RequestMapping with Mode should be unique with respect to controller class.

```
@GetMapping(value={"/report"})
public  String showReport(Map<String,Object> map) {
            System.out.println("TestController.showReport(-)");
            return "display";
}
@GetMapping(value={"/report"})
public  String showReport1(Map<String,Object> map) {
            System.out.println("TestController.showReport1(-)");
            return "display";
}
========================================================================
@GetMapping
public   String  showHome() {
      System.out.println("TestController.showHome()");  =====> urlPattern :: / and
RequestMode :: GET
      return "home";
}

@GetMapping
public   String  showHome1() {
      System.out.println("TestController.showHome()"); =====> urlPattern :: / and
RequestMode :: GET
      return "home";
}

=====================================================================
```

g. In SpringMVC max two methods can be taken without requestpath(One with GET request and another one with POST request)

```
@PostMapping
public   String  showHome() {
      System.out.println("TestController.showHome()");
      return "home";
}
@GetMapping
public   String  showHome1() {
      System.out.println("TestController.showHome1()");
      return "home";
```

```
}
```
h. If two controller class have two handler methods with same request type and request mode then we need to differentiate them at the
   Controller level using "global request path"

```
@Controller
public class DemoController {
      @GetMapping(value={"/report"})
      public  String generateReport(Map<String,Object> map) {
            return "display";
      }
}

@Controller
public class TestController {
      @GetMapping(value={"/report"})
      public  String generateReport(Map<String,Object> map) {
            return "display";
      }
}
```

Solution
========
```
@Controller
@RequestMapping("/demo")//global requestpath
public class DemoController {
      @GetMapping(value={"/report"})
      public  String generateReport(Map<String,Object> map) {
            return "display";
      }
}

@Controller
@RequestMapping("/test")//global requestpath
public class TestController {
      @GetMapping(value={"/report"})
      public  String generateReport(Map<String,Object> map) {
            return "display";
      }
}
```

requestURL :: http://localhost:9999/ThirdApp/demo/report ===>
DemoController.generateReport()
requestURL :: http://localhost:9999/ThirdApp/test/report ===>
TestController.generateReport()

j. We cannot take ServletContext,ServletConfig object as a parameter of handler method becoz they are global objects which go for
   @Autowired injections more over they are not valid parameter types for handler methods,but we can take HttpSession type
   parameter in handler methods.

```
@Controller
@RequestMapping("/demo")
public class DemoController {
      @GetMapping(value={"/report"})
      public  String generateReport(Map<String,Object> map,HttpSession ses) {
            System.out.println("DemoController.generateReport(-)");
            System.out.println("Session Id:"+ses.getId());
```

```
            return "display";
        }
}
```