*Spring MVC Framework provides decoupled way of developing web applications. With simple concepts like Dispatcher Servlet, ModelAndView and View Resolver, it makes it easy to develop web applications.*

In short, DispatcherServlet is used following things in Spring MVC  
- receives all request as Front Controller  and provides a single entry point to the application  
- mapping requests to correct [Spring MVC](https://javarevisited.blogspot.sg/2013/07/role-based-access-control-using-spring-security-ldap-authorities-mapping-mvc.html) controller  
- Consulting ViewResolvers to find correct View  
- forwarding request to chosen View for rendering  
- returning the response to the client  
- creates web-context to initialize the web specific beans e.g. [controllers](http://javarevisited.blogspot.sg/2011/09/spring-interview-questions-answers-j2ee.html), view resolvers and handler mapping

The DispatcherServlet is one of the important components of Spring MVC web framework and acts as a Front Controller. Similar to other Java web framework, Spring MVC also uses a front controller to receive all incoming request and delegates to other components for further processing e.g. Spring MVC controllers which are annotated using @Controller annotation and ViewResolvers e.g InternalResourceViewResolver class.

A Front Controller (see Patterns of Enterprise Application Architecture) is a common pattern in web application and used to receive request and delegate to other components in the application for actual processing. The DispatcherServlet is a front controller e.g. it provides a single entry point for a client request to Spring MVC web application and forwards request to Spring MVC controllers for processing.

The DispatcherServlet is like any other Servlet class and it has to be declared inside the deployment descriptor or web.xml file as shown be

<servlet>

<servlet-name>dispatcher</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<init-param>

<param-name>contextConfigLocation</param-name>

<!--Defaults to WEB-INF\dispatcher-servlet.xml -->

<param-value>classpath:mvc-config.xml</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>dispatcher</servlet-name>

<url-pattern>/\*</url-pattern>

</servlet-mapping>

CMI:

<servlet>

<servlet-name>RemoteController</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>RemoteController</servlet-name>

<url-pattern>/remote/\*</url-pattern>

</servlet-mapping>

I haven't shown here but, DispatcherServlet is also usually preloaded using the load-on-startup tag of the deployment descriptor. You can give zero or positive value on this tag to pre-load a servlet, otherwise, the servlet will only be loaded when a request will arise.

If your servlet does a lot of job on initialization e.g. DispatcherServlet which initializes all the beans declared in its web context e.g. controllers, view resolvers, and mapping handlers then it could slow down the response time.

Btw, it's not the only way to declare DispatcherServlet in Spring MVC. From Spring 3.2 and Servlet 3.0 specification, you can programmatically declare DispatcherServlet using ServletContainerInitializer interface. This is a Servlet 3,0 feature which allows Servlet 3.0 compliant containers e.g. Tomcat 7 or higher to scan and load any class which implements this interface.

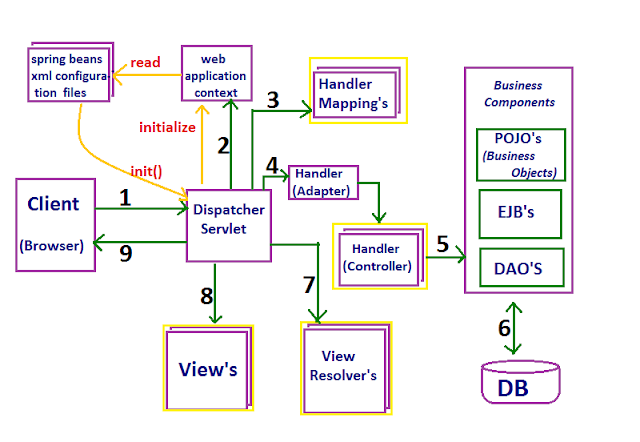
Spring provides an implementation of this interface as SpringServletContainerInitializer and a convenient class called AbstractAnnotationConfigDispatcherServletInitialzer in Spring 3.2 to configure DispatcherServlet without deployment descriptor.

This class implements SpringServletContainerInitializer hence automatically picked by Servlet 3.0 compliant containers. See Spring in Action 4th Edition to know more about configuring DispatcherServlet using Java Configuration.

**How dispatcher servlet works internally?**

As I said, DispatcherServlet wears many hats in Spring. It acts as a front controller and provides a single entry point for the application. It then uses handler mappings and handler adapters to map a request to the [Spring MVC controllers](http://javarevisited.blogspot.sg/2012/05/what-is-bean-scope-in-spring-mvc.html#axzz4jWEJmi6S). It uses @Controller and @RequestMapping annotation for that purpose.  
  
Once the request is processed by Spring MVC controller, it returns a logical view name instead of the view. Though, you can even configure Controler's handler methods to not return any View name by declaring return type as void. You can even use @ResponseBody annotation in the case of REST to directly write the output to the HTTP response body. See [REST with Spring](http://www.baeldung.com/rest-with-spring-course?utm_source=javarevisited&utm_medium=web&utm_campaign=rws&affcode=22136_bkwjs9xa) course by Eugen to learn more about developing RESTful web services using Spring MVC.  
  
When[DispatherServlet](http://javarevisited.blogspot.sg/2016/01/solving-javalangclassnotfoundexception-org.springframework.web.servlet.DispatcherServlet.html" \t "_blank) receives view name, it consults the ViewResolver to find the right view. There is a chain of ViewResolver is maintained at Spring MVC framework. They try to resolve the logical view name into a Physical resource e.g. a JSP page or a FreeMaker or Velocity template.  
  
The **ViewResolver** are invoked in an order, if first in the chain not able to resolve the view then it returns null and next ViewResolver in the chain is consults. Once the right view is found, DispatcherServlet forwards the request along with Model data to the View for rendering e.g. a [JSP page](http://javarevisited.blogspot.sg/2017/01/best-books-to-learn-servlet-and-jsp.html).

By default, DispatcherServlet uses **InternalResourceViewResolver** which uses prefix and suffix to convert a logical view name e.g. "home" to /WEB-INF/home.jsp. The View interface also has getContentType() method, which returns content type the view produces (JstlView has text/html). This is usually the default content type for requests handled by the dispatcher servlet in Spring.  
  
Here is a nice diagram which explains how DispatcherServlet works internally in Spring MVC



View Resolver

View Resolver pattern is a J2EE pattern that enables a web application to dynamically select its view technology. For example, HTML, JSP, Tapestry, JSF, XSLT etc. In this pattern, View resolver holds mapping of different views, controller return name of the view, which is then delegated to the View Resolver for selecting an appropriate view. Spring MVC framework supplies inbuilt view resolver for selecting views.

CMI:

<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">

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

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

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

</bean>

1. When u want to return the view name then Model is one of the arguments in the mapping method

Ex:

@RequestMapping(value = "/SAS-US-CMBS-DealLoanDetails", method = RequestMethod.POST)

**public** String dealLoanDetailValues(@RequestParam("dealId") String dealId,

@RequestParam("cmpDealId") String cmpDealId, @RequestParam("srcCd") String srcCd,

@RequestParam("dealPerfVerNum") String dealPerfVerNum, @RequestParam("asOfDate") String asOfDate,

@RequestParam("maxResults") String maxResults, @RequestParam("newIssuanceFlag") String newIssuanceFlag,

@RequestParam("dealName") String dealName, HttpServletRequest req, HttpServletResponse res, Model model) {

return “someViewName”;

}

1. When u want to return modelAndView then

Ex:

@RequestMapping("/SAS-US-CMBS-DealOverview")

**public** ModelAndView dealOverview(@RequestParam("drmId") String drmId,

@RequestParam("cmRunId") String cmRunId,@RequestParam("cfmRunId") String cfmRunId,

HttpServletRequest req,HttpServletResponse res) {

ModelAndView mv = **new** ModelAndView("DRM/dealOverview");

Return mv;

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

ModelAnd View

Holder for both Model and View in the web MVC framework. Note that these are entirely distinct. This class merely holds both to make it possible for a controller to return both model and view in a single return value.

Represents a model and view returned by a handler, to be resolved by a DispatcherPortlet. The view can take the form of a String view name which will need to be resolved by a ViewResolver object; alternatively a view object can be specified directly. The model is a Map, allowing the use of multiple objects keyed by name.

Difference Between @RequestParam and @PathVariable in Spring MVC.

@RequestParam and @PathVariable annotations are used for accessing the values from the request.

The primary difference between @RequestParam and @PathVariable is that @RequestParam used for accessing the values of the query parameters where as @PathVariable used for accessing the values from the URI template.

Explain Interceptors in Spring MVC framework.

Handler interceptors are required when specific functionality need to be applied to certain requests. Handler Interceptors should implement the interface HandlerInterceptor and override the below methods.

**preHandle**is called before the actual handler is executed;

**postHandle** is called after the handler is executed;

**afterCompletion** is called after the request is complete.

What are the different Controller implementations in Spring MVC framework?

* Controller,
* AbstractCommandController,
* SimpleFormController,
* WizardFormController,
* and MultiActionController.

CMI:

What is a MultipartResolver in Spring MVC framework?

MultipartResolver interface is used for uploading files; CommonsMultipartResolver and StandardServletMultipartResolver are 2 implementations provided by spring framework to facilitate file uploading. By default there are no multipart resolvers configured, to enable it define a bean named 'multipartResolver' with type as MultipartResolver in spring bean configurations.

Once configured, any multipart request will be resolved by the configured MultipartResolver and pass on a wrapped HttpServletRequest. Then it is used in a controller class to get the file and process it.

<!-- Configure the multipart resolver -->

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

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

<property name="maxUploadSize" value="10240000000"/>

<property name="defaultEncoding" value="UTF-8" />

</bean>

In controller

public @ResponseBody String uploadAnalysisDeal(TemplateUploadVO uploadItem, BindingResult bResult, HttpServletRequest req, HttpServletResponse res) {}

In TemplateUploadVO

private CommonsMultipartFile templateFile;

How to configure DispatcherServlet without web.xml in Spring MVC?

Create a class implementing WebApplicationInitializer interface and override onStartup() method. In the method, we can register annotation based application configuration class, servlet and mappings, listener etc.

How do I return a string from the Spring MVC controller without a view?

Set the return type of the method as String and mark the method with @ResponseBody annotation.

**@RequestMapping**(value="/returnHelloWorld", method=GET)

**@ResponseBody**

**public** String **returnHelloMethod**() {

**return** "Hello world!";

}

Explain @ResponseBody annotation in Spring MVC.

When a controller method is marked with @ResponseBody, spring deserializes the returned value of the method and writes it directly to the Http Response automatically.

The return value of the method constitute the body of the HTTP response and not placed in a Model, or interpreted as a view name.

Explain @RequestBody annotation in Spring MVC.

Spring automatically converts the content of the incoming request body to the parameter object when annotated with the @RequestBody annotation.

**@ResponseBody** **@RequestMapping**("/getUserInfo")

**public** String **getUserInformation**(**@RequestBody** UserDetails user){

**return** user.getFirstName() + " " + user.getLastName();

}

How do I configure JNDI DataSource in Spring Web Application?

To use the servlet container configured JNDI DataSource, we need to wire it in the spring bean configuration file and then inject it to spring beans as dependency. Once done, we can use it with JdbcTemplate to perform database operations.

< bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">

< property name="jndiName" value="java:comp/env/jdbc/MySQLDB"/>

< /bean>

CMI:

<jee:jndi-lookup id=*"dataSourceSfdlexp"* jndi-name=*"jdbc/dataCmbsSfdlexp"*/>

<bean id=*"hibernatePropertiesSfdlexp"*

class=*"org.springframework.beans.factory.config.PropertiesFactoryBean"*>

<property name=*"properties"*>

<props>

<prop key=*"hibernate.dialect"*>org.hibernate.dialect.Oracle10gDialect</prop>

<prop key=*"hibernate.connection.autocommit"*>true</prop>

<prop key=*"hibernate.connection.release\_mode"*>after\_transaction</prop>

<prop key=*"hibernate.default\_schema"*>SFDLEXP</prop>

<prop key=*"hibernate.show\_sql"*>false</prop>

<prop key=*"jadira.usertype.autoRegisterUserTypes"*>true</prop>

<prop key=*"jadira.usertype.databaseZone"*>jvm</prop>

<prop key=*"jadira.usertype.javaZone"*>jvm</prop>

<prop key=*"hibernate.format\_sql"*>false</prop>

<prop key=*"hibernate.query.factory\_class"*>org.hibernate.hql.internal.classic.ClassicQueryTranslatorFactory</prop>

<prop key=*"hibernate.jdbc.batch\_size"*>300</prop>

<prop key=*"hibernate.jdbc.fetch\_size"*>2000</prop>

<!--

<prop key="hibernate.cache.provider\_class">org.hibernate.cache.EhCacheProvider</prop>

<prop key="hibernate.cache.use\_query\_cache">true</prop>

<prop key="hibernate.cache.use\_second\_level\_cache">true</prop>

<prop key="hibernate.cache.region.factory\_class">net.sf.ehcache.hibernate.EhCacheRegionFactory</prop>

<prop key="hibernate.cache.provider\_configuration\_file\_resource\_path">ehcache.xml</prop>

<prop key="hibernate.cache.disk-persistent">true</prop>

-->

</props>

</property>

</bean>

<!-- <bean id="dataSourceSfdlexp2" class="org.springframework.jndi.JndiObjectFactoryBean">

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

<property name="jndiName" value="jdbc/dataCmbsSfdlexp" />

</bean> -->

<!-- Enable annotation style of managing transactions -->

<tx:annotation-driven transaction-manager=*"transactionManagerSfdlexp"*/>

<bean id=*"sessionFactory"*

class=*"org.springframework.orm.hibernate5.LocalSessionFactoryBean"*>

<property name=*"dataSource"* ref=*"dataSourceSfdlexp"*/>

<property name=*"packagesToScan"*>

<list>

<value>com.sp.sf.sas.us.dto.\*\*</value>

<value>com.sp.sf.sas.us.domain</value>

<value>com.sp.sf.sas.us.yieldAnalysis</value>

</list>

</property>

<property name=*"mappingLocations"*>

<list>

<value>classpath:cmbs-sql-named-queries.xml</value>

</list>

</property>

<property name=*"hibernateProperties"* ref=*"hibernatePropertiesSfdlexp"*></property>

</bean>

<bean id=*"transactionManagerSfdlexp"*

class=*"org.springframework.orm.hibernate5.HibernateTransactionManager"*>

<property name=*"sessionFactory"* ref=*"sessionFactory"*/>

</bean>

## Spring Interceptor – HandlerInterceptor

HandlerInterceptor interface can be used in spring mvc application to pre-handle and post-handle web requests that are handled by Spring MVC controllers. These handlers are mostly used to manipulate the model attributes returned/submitted before it is passed to the views/controllers.

A handler interceptor can be registered for particular URL mappings so that it only intercepts requests mapped to certain URLs. Each handler interceptor must implement the HandlerInterceptor interface, which contains three callback methods for you to implement: preHandle(), postHandle() and afterCompletion().

Spring **HandlerInterceptor** declares three methods based on where we want to intercept the HTTP request.

1. **boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)**: This method is used to intercept the request before it’s handed over to the handler method. This method should return ‘true’ to let Spring know to process the request through another spring interceptor or to send it to handler method if there are no further spring interceptors.

If this method returns ‘false’ [Spring framework](https://www.journaldev.com/16922/spring-framework) assumes that request has been handled by the spring interceptor itself and no further processing is needed. We should use response object to send response to the client request in this case.

Object handler is the chosen handler object to handle the request. This method can throw Exception also, in that case [Spring MVC Exception Handling](https://www.journaldev.com/2651/spring-mvc-exception-handling-controlleradvice-exceptionhandler-handlerexceptionresolver) should be useful to send error page as response.

1. **void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView)**: This HandlerInterceptor interceptor method is called when HandlerAdapter has invoked the handler but DispatcherServlet is yet to render the view. This method can be used to add additional attribute to the ModelAndView object to be used in the view pages. We can use this spring interceptor method to determine the time taken by handler method to process the client request.
2. **void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex)**: This is a HandlerInterceptor callback method that is called once the handler is executed and view is rendered.

If there are multiple spring interceptors configured, preHandle() method is executed in the order of configuration whereas postHandle() and afterCompletion() methods are invoked in the reverse order.

Ex:

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.slf4j.Logger;

import org.slf4j.LoggerFactory;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.web.servlet.handler.HandlerInterceptorAdapter;

public class RequestProcessingTimeInterceptor extends HandlerInterceptorAdapter {

private static final Logger logger = LoggerFactory

.getLogger(RequestProcessingTimeInterceptor.class);

@Override

public boolean preHandle(HttpServletRequest request,

HttpServletResponse response, Object handler) throws Exception {

long startTime = System.currentTimeMillis();

logger.info("Request URL::" + request.getRequestURL().toString()

+ ":: Start Time=" + System.currentTimeMillis());

request.setAttribute("startTime", startTime);

//if returned false, we need to make sure 'response' is sent

return true;

}

@Override

public void postHandle(HttpServletRequest request,

HttpServletResponse response, Object handler,

ModelAndView modelAndView) throws Exception {

System.out.println("Request URL::" + request.getRequestURL().toString()

+ " Sent to Handler :: Current Time=" + System.currentTimeMillis());

//we can add attributes in the modelAndView and use that in the view page

}

@Override

public void afterCompletion(HttpServletRequest request,

HttpServletResponse response, Object handler, Exception ex)

throws Exception {

long startTime = (Long) request.getAttribute("startTime");

logger.info("Request URL::" + request.getRequestURL().toString()

+ ":: End Time=" + System.currentTimeMillis());

logger.info("Request URL::" + request.getRequestURL().toString()

+ ":: Time Taken=" + (System.currentTimeMillis() - startTime));

}

}

applicationContext.xml

<interceptors>

<interceptor>

<mapping path="/home" />

<beans:bean class="com.journaldev.spring.RequestProcessingTimeInterceptor"></beans:bean>

</interceptor>

</interceptors>

Use of BindingResult interface in Spring MVC.

Use a BindingResult object as an argument to the validate method of a Validator inside a Controller and the BindingResult object will hold the validation errors.

validator.validate(modelObject, bindingResult);

**if** (bindingResult.hasErrors()) {

// Handle error

}

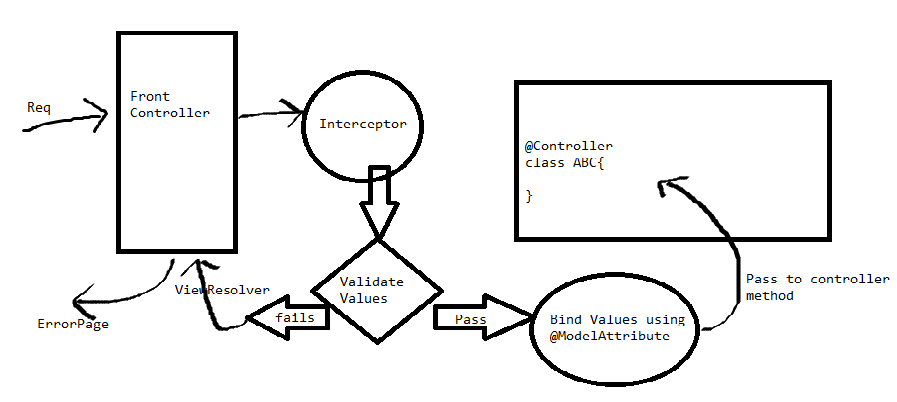
[BindingResult] is Spring’s object that **holds the result of the validation and binding and contains errors that may have occurred.** The BindingResult must come right after the model object that is validated or else Spring will fail to validate the object and throw an exception.

When Spring sees @Valid, it tries to find the validator for the object being validated. Spring automatically picks up validation annotations if you have “annotation-driven” enabled. Spring then invokes the validator and puts any errors in the BindingResult and adds the BindingResult to the view model.

Well its a sequential process. The Request first treat by FrontController and then moves towards our own customize controller with @Controller annotation.

but our controller method is binding bean using modelattribute and we are also performing few validations on bean values.

so instead of moving the request to our controller class, FrontController moves it towards one interceptor which creates the temp object of our bean and the validate the values. if validation successful then bind the temp obj values with our actual bean which is stored in @ModelAttribute otherwise if validation fails it does not bind and moves the resp towards error page or wherever u want.

[](https://i.stack.imgur.com/fdlxU.png)

What is resource-ref in web.xml used for?

It resource reference to an object factory for resources such as a JDBC DataSource, a JavaMail Session, or custom object factories configured into Tomcat or any other application server.

How JNDI lookup differs between Tomcat and weblogic server?

In Weblogic, the JNDI lookup for "dsJDBC" is just "dsJDBC" whilst in Tomcat, it accepts only the the formal format "java:comp/env/jdbc/dsJDBC".

In Tomcat, JndiLocatorSupport has a property resourceRef. When setting this true, "java:comp/env/" prefix will be prepended automatically.