Spring MVC Workflow

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution.

```

DispatcherServlet :

```

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

* The DispatcherServlet of Spring Web MVC framework is an implementation of FrontController and is a Java Servlet component. i.e., it is a servlet front for spring Web Mvc application.
* DispathcherServlet is the FrontController classes that receives all incoming HTTP client request for the spring Web mvc appliaction.
* DispatcherServlet is responsible for initializing spring web mvc framework for our application, and is a servlet implemented as a sub-type of HttpServlet just like any other Servlet.
* DispatcherServlet also required to be configured in our web-application like any other Servlet i.e., web application deployment descriptor(web.xml)

DispatcherServlet

``` xml

<servlet>

<servlet-name>ds</servlet-name>

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

<load-on-startup>0</load-on-startup>

</servlet>

```

We can configure the initialization parameters to alter the behavior of the DispatcherServlet with respect to locating the spring Beans XML configuration files and initializing the application context.

```

Initialization Parameters of DispatcherServlet:

```

1. contextClass
2. namespace
3. contextConfigLocation
4. publishContext
5. publishEvents
6. detectAllHandlerMappings
7. detectAllHandlerAdapters
8. detectAllHandlerExceptionResolvers
9. detectAllViewResolvers
10. cleanupAfterInclude

``` xml

<servlet>

<servlet-name>ds</servlet-name>

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

<init-param>

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

<param-value>

/WEB-INF/applicationContext.xml

/WEB-INF/myconfigs/applicationControllers.xml

</param-value>

</init-param>

<load-on-startup>0</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>ds</servlet-name>

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

</servlet-mapping>

```

* All the request for this context whose servletPath ends with **.spring** extension are displayed by the web container to the DispatcherServlet.
* It is not mandatory to configure only .spring, instead of configure any other extension.

Spring Web MVC framework does not force to use specific extensions or patterns in URL's;  
Spring frame work is not bonded to match the handlers only based on URLs.Alternatively we can define a mapping based on parameters or HTTPSession state etc.,   
This future is not supported by most of Web MVC framework which includes Struts.

### DispatcherServlet Initialization Stage :

* While initializing the DispatcherServlet it creates WebApplicationContext implementing class instance, either user configured custom context class (or) the default XmlWebApplicationContext.
* The WebApplicationContext is responsible to locate the Spring Beans XML configuration file, then read, validate the configurations and load the details into configuration objects.
* The DispatcherServlet uses the WebApplicationContext for accessing various framework objects that are used to execute the WebApplicationContext.

Spring WEB MVC Framework special objects :

1.MultipartResolver

2.LocaleResolver

3.ThemeResolver

4.HandlerMapping

5.HandlerAdapter

6.HandlerExceptionResolver

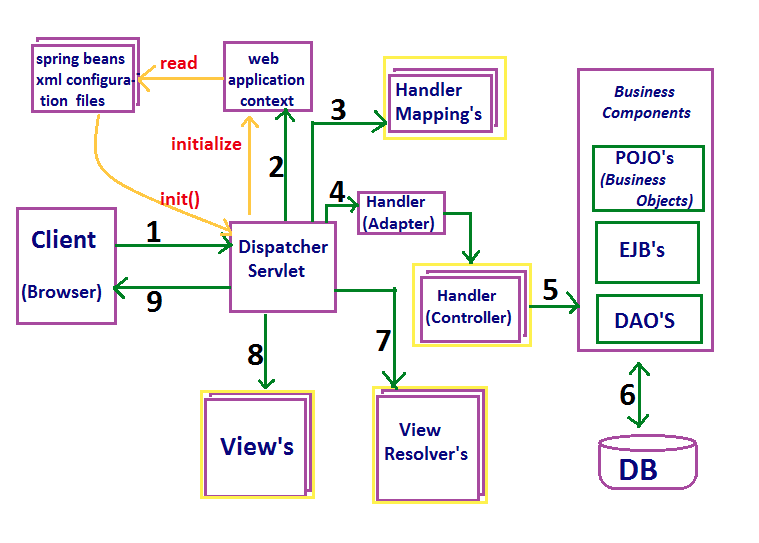
7.RequestToViewNameTranslator

8.ViewResolver

* All the special frame work objects initialized by the DispatcherServlet using the WebApplicationContext and based on the detectAllXXX initialization parameters configured to this servlet.
* Once if all these objects are initialized successfully, the DispatcherServlet instance is put into service, which provides an entry point to serve the client requests using the special frame work objects.

## Spring WEB MVC Request Processing Work flow :

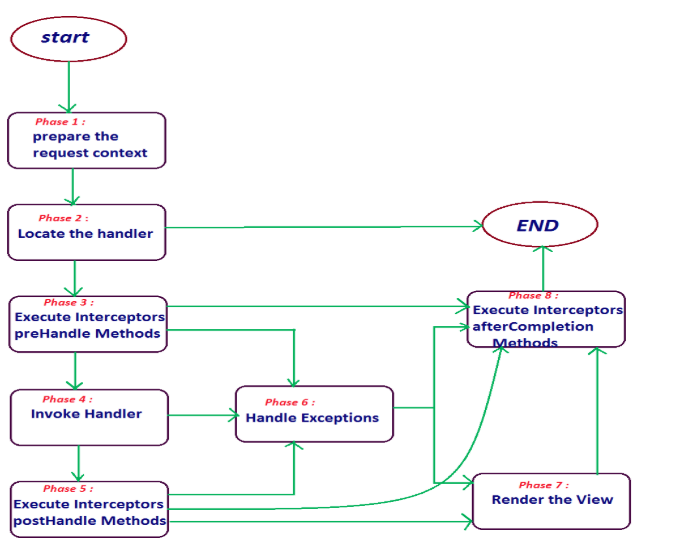
* According to spring MVC , DispatcherServlet is the front controller for the spring Web MVC application, providing a centralized access for various requests to the application and collaborating with various other objects to complete the request handling and present the response to client.
* DispatcherServlet uses the WebApplicationContext object to locate the various objects configured in the Spring Bean XML configuration file.
* The WebApplicationContext is instantiated and initialized as a part of the DispatcherServlet's initialization process.
* The WebApplicationContext object is responsible to locate the spring beans XML configuration file and load its details to prepare the context for handling the requests.



Now, when a client request is given to DispatcherServlet, it performs the following operations:

Types of Phases :

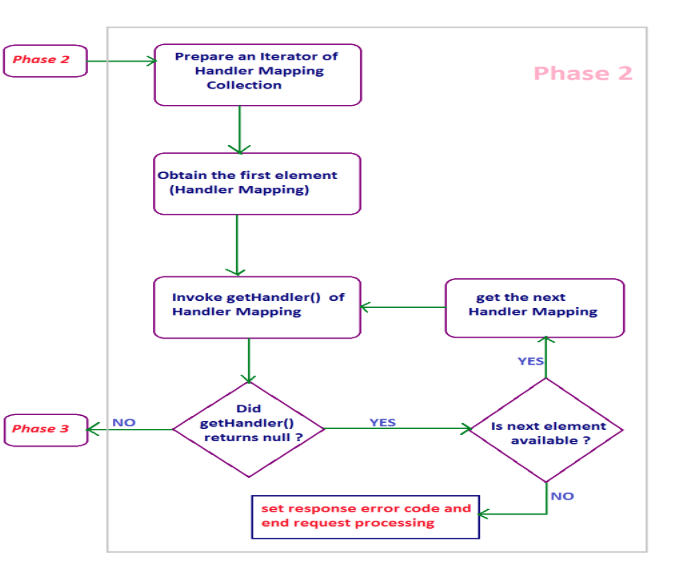
1. Prepare the request context
2. Locate The Handler
3. Execute Interceptors preHandle methods
4. Invoke Handler
5. Execute Interceptors postHandle methods
6. Handle Exceptions
7. Render the View
8. Execute Interceptors afterCompletion methods



### Phase 1 : Prepare the request context

1. DispatcherServlet prepare the request context, by setting the framework objects into the request scope.
2. Here framework objects are WebApplicationContext, LocaleResolver, ThemeResolver and ThemeSource.
3. These objects are set into the request scope to make them available to handler and view objects.
4. so that the handler or view objects can use these objects to communicate with the framework and collect runtime details.
5. Apart from preparing and setting the framework objects into the request scope, DispatcherServlet resolves the request using MultipartResolver , so that if the request contains contains multi part data , then it wraps the request in a MultipartHttpRequest type object.
6. In case , if there is any problem in this process the request processing is terminated by throwing an Exception.
7. Once if this process is done successfully the request process workflow continues to the next phase, i.e., Locate the Handler.

### Phase 2 : Locate The Handler



1. After preparing the request context , the DispatcherServlet locates handler that can handle this request.
2. The DispatcherServlet uses the registered HandlerMapping's and collects the HandlerExecutionChain object.
3. The HandlerExecutionChain object encapsulates the HandlerIntercepter's and the Handler object (i.e., controller).
4. Preparing an Iterator object of the Collection, storing the HandlerMapping objects.
5. This Collection object is created while the DispatcherServlet is being initialized, i.e., in its initialization phase.
6. Thereafter, the first element is received from the iterator. This can't be an empty collection since if there is no HandlerMapping configured in the context the DispatcherServlet uses BeanNameUrlHandlerMapping as default HandlerMapping.
7. Invoke the getHandler() of the current HandlerMapping object in the iteration.

* If getHandler() returns null, get the next element is available the next HandlerMapping. Otherwise set response error code(i.e., HttpServletResponse\_SC\_NOT\_FOUND) and terminate request processing.
* If getHandler() returns a valid HandlerExecutionChain object reference then the control delegates to next phase.(i.e., Execute Interceptors preHandle methods )

##### The HandlerMappings :

1. The HandlerMapping is responsible for mapping the incoming request to the handler that can handle the request.
2. As discussed in the above section, When the DispatcherServlet receives the request it delegates the request to the HandlerMapping, which identifies the appropriate HandlerExecutionChain that can handle the request.
3. The Spring Web MVC framework provides customizable navigation strategies. Spring provides built-in navigation strategies as determining the handler based on the request URL mapping , which is again based on the bean name.
4. A part from the built-in navigation strategies, Spring allows system-specific built strategy, which can be done by writing a class implementing the HandlerMapping interface.

The Spring built-in HandlerMapping implementations are :

1. BeanNameUrlHandlerMapping
2. SimpleUrlHandlerMapping
3. ControllerClassNameHandlerMapping
4. CommonsPathMapHandlerMapping

##### BeanNameUrlHandlerMapping

* The org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping is one of the implementation of HandlerMapping interface.
* This implementation defines the navigation strategy that maps the request URL's servlet-path to the bean names.
* This handler mapping strategy is very simple but powerful.

The BeanNameUrlHandlerMapping is the default handler mapping when no handler mapping is configured in the application context.

**The BeanNameUrlHandlerMapping explicitly configure in 2 cases :**

1. When we want to configure multiple handler mapping
2. When we want to configure handler interceptors

ds-servlet.xml

``` xml

<beans>

<bean id="handlerMapping"

class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

<bean name="/addEmployee.spring" class="com.spring.AddEmployeeController">

<!-- set the dependencies -->

</bean>

<bean name="/removeEmployee.spring" class="com.spring.RemoveEmployeeController">

<!-- set the dependencies -->

</bean>

</beans>

```

##### SimpleUrlHandlerMapping

* The org.springframework.web.servlet.handler.SimpleUrlHandlerMapping is one of the implementation of HandlerMapping interface.
* This implementation defines the navigation strategy that maps the request URL's servlet-path to the mapping configured. i.e., It locates the handler(controller) by matching the request URL's servlet-path with the key of the given properties or Map.
* The SimpleUrlHandlerMapping supports the configure 2 types :

1. bean names
2. bean instances

``` xml

<beans>

<bean id="addEmp" class="com.spring.AddEmployeeController">

<!-- configure dependencies -->

</bean>

<bean id="removeEmp" class="com.spring.RemoveEmployeeController">

<!-- configure dependencies -->

</bean>

<bean id="searchEmp" class="com.spring.SearchEmployeeController">

<!-- configure dependencies -->

</bean>

<bean id="handlerMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="mappings">

<props>

<prop key="/addEmployee.spring">addEmp</prop>

<prop key="/removeEmployee.spring">removeEmp</prop>

<prop key="/searchEmployee.spring">searchEmp</prop>

</props>

</property>

</bean>

</beans>

```

This configuration is suitable for configuring non-singleton beans

**OR**

``` xml

<beans>

<bean id="addEmp" class="com.spring.AddEmployeeController">

<!-- configure dependencies -->

</bean>

<bean id="removeEmp" class="com.spring.RemoveEmployeeController">

<!-- configure dependencies -->

</bean>

<bean id="searchEmp" class="com.spring.SearchEmployeeController">

<!-- configure dependencies -->

</bean>

<bean id="handlerMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="mappings">

<map>

<entry key="/addEmployee.spring">

<ref local="addEmp"/>

</entry>

<entry key="/removeEmployee.spring">

<ref local="removeEmp"/>

</entry>

<entry key="/searchEmployee.spring">

<ref local="searchEmp"/>

</entry>

</map>

</property>

</bean>

</beans>

```

This configuration is suitable for configuring singleton beans

##### ControllerClassNameHandlerMapping

1. The org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping is one of the implementation of HandlerMapping interface.
2. This handler mapping implementation is newly introduced in spring 2.0
3. The ControllerClassNameHandlerMapping follows a simple convention for generating URL path mappings.
4. The convention for simple Controller implementations is to take the short names of the controller class.

Ex : If the controller class name is **com.emp.AddEmployeeController** then the path is **addemployee\***

``` xml

<beans>

<bean id="handlerMapping"

class="org.springframework.web.servlet.mvc.

support.ControllerClassNameHandlerMapping"/>

<bean id="addEmp" class="com.spring.AddEmployeeController">

<!-- configure dependencies -->

</bean>

<bean id="removeEmp" class="com.spring.RemoveEmployeeController">

<!-- configure dependencies -->

</bean>

<bean id="searchEmp" class="com.spring.SearchEmployeeController">

<!-- configure dependencies -->

</bean>

</beans>

```

##### CommonsPathMapHandlerMapping

* The **org.springframework.web.servlet.handler.CommonsPathMapHandlerMapping** is one of the implementation of HandlerMapping interface.
* The **org.springframework.web.servlet.handler.CommonsPathMapHandlerMapping** is designed to recognize Commons attributes meta data attributes of type PathMap defined in the application controller and automatically wires them in to the current DiapatcherServlet's Web-application context.
* To use this HandlerMapping the controller class must have a class level metadata of the form **@org.springframework.web.servlet.handler.commonsattribures.PathMap("/mypath.spring")**.
* We can configure multiple path maps for a single controller.

``` xml

@org.springframework.web.servlet.handler.commonsattribures.PathMap(

"/mypath.spring")

public class FirstController implements Controller {

public ModelAndView handleRequest(HttpServletRequest request,

HttpServletResponse response)throws Exception {

System.out.println("we are in handleRequest()");

ModelAndView mav=null;

return mav;

}

```

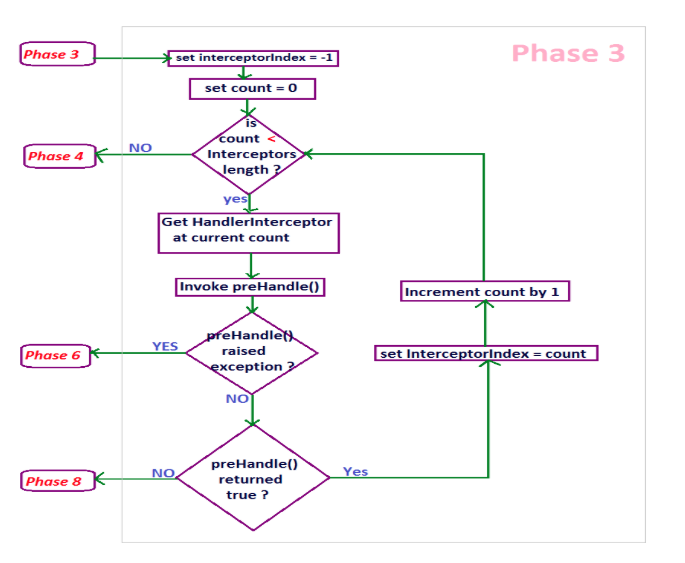
Note : To use commons attributes path mapping , we must compile application classes with Commons Attributes, and run the Commons attributes indexer tool on the application classes, which must be in a Jar rather than in WEB-INF/classes as an individual classes.

**Configuring multiple handler mappings :**

* We can configure multiple handler mappings in an application context.
* In such a case we need to configure an additional property 'order' that takes 'int' value on each of the handler mapping.

### Phase 3 : Execute Interceptors preHandle methods

1. After successfully locating the HandlerExcecutionChain the DispatcherServlet executes the HandlerInterceptors described by the HandlerExecutionChain returned by the HandlerMapping.
2. The HandlerInterceprors gives us an opportunity to add common pre and post-processing behavior without needing to modify each handler implementation.
3. The HandlerInterceptors is basically similar to a Servlet Filter(2.3v) .



1. The HandlerInterceptors can be used for implementing pre-processing aspects, for example , for authorization checks, or common handler behavior like locale or theme changes.

Using the HandlerInterceptor includes the following 2 steps :

1. Write an HandlerInterceptor implementation
2. Configure the interceptor

**Write an HandlerInterceptor implementation :**

* The HandlerInterceptor can be used for implementing pre-processing aspects, for example, for authorization checks, or common handler behavior like locale or theme changes.
* The Spring framework includes built-in HandlerInterceptor's implementing the most common pre-processing concerns like locale and theme change.

The following are the built-in HandlerInterceptor implementations

* org.springframework.web.servlet.i18n.LocaleChangeInterceptor
* org.springframework.web.servlet.theme.ThemeChangeInterceptor

Apart from the built-in HandlerInterceptors we can write HandlerInterceptor implementations encapsulating the custom pre- and post-processing logics.

``` xml

package com.spring;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.HandlerInterceptor;

import org.springframework.web.servlet.ModelAndView;

public class MyHandlerInterceptor implements HandlerInterceptor {

public boolean preHandle(HttpServletRequest request,

HttpServletResponse response,

Object handler) throws Exception {

System.out.println("preHandle() executes at phase 3");

/\*

Do some preprocessing like check the security etc

return boolean value accordingly,

if we want to continue the interceptor chain returns true

\*/

return false;

}

public void postHandle(HttpServletRequest request,

HttpServletResponse response,

Object handler, ModelAndView mav) throws Exception {

System.out.println("afterCompletion() executes at phase 5");

//Do some postprocessing

}

public void afterCompletion(HttpServletRequest request,

HttpServletResponse response,

Object handler, Exception ex)throws Exception {

System.out.println("afterCompletion() executes at phase 5");

//Do finalizations

}

}

```

**Configuring the interpretor :**

``` xml

<beans>

<bean id="myInterceptor" class="com.spring.MyHandlerInterceptor"/>

<bean id="handlerMapping"

class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">

<property name="interceptors">

<list>

<ref local="myInterceptor"/>

</list>

</property>

</bean>

</beans>

```

### Phase 4 : Invoke Handler

1. In this phase of Spring web MVC request processing workflow the DispatcherServlet delegates the request to the handler that is located by the HandlerMapping in Phase 2.
2. DispatcherServlet uses HandlerAdapter to delegate the request to the handler located to handle this request.

**step 1 : Prepare a iterator of HandleAdapter collection :**

1. In this Phase workflow starts with preparing an iterator of the collection representing the HandlerAdapter objects configured in the application.
2. The handlerAdapters collection is initialized in the initialization phase of the DispatcherServlet where it finds all HandlerAdapters in the ApplicationContext.
3. If no HandlerAdapter beans are defined in the application then the default is considered as SimpleControllerHandlerAdapter. Thus the handlerAdapters collection contains at least one element.

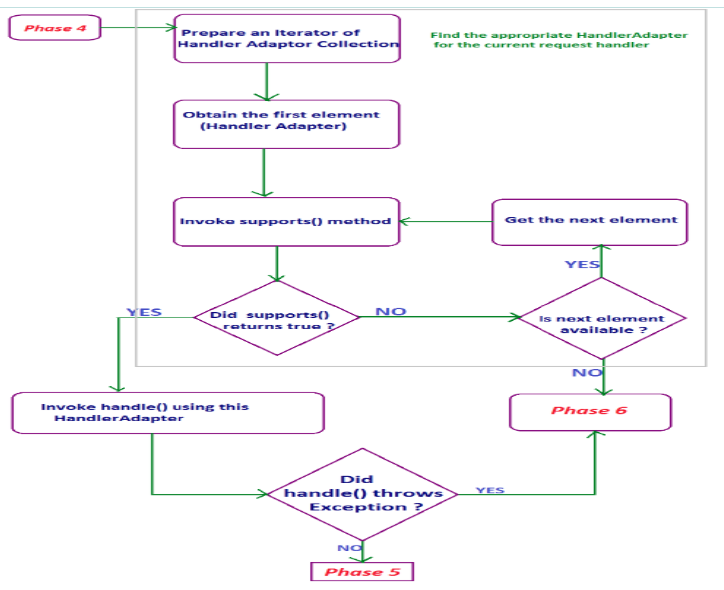
**step 2 : Get HandlerAdapter :**

1. In this step the workflow obtains the next element from the iterator in step 1 and casts it into HandlerAdapter type reference.

**step 3 : Find is HandlerAdapter compatible :**

1. After getting the HandlerAdapter of the current iteration the workflow continues with finding whether this HandlerAdapter is suitable for the handler located in phase 2.
2. This is done by invoking supports() method on the HandlerAdapter.
3. If it returns "true" then it indicates that this HandlerAdapter supports the handler. In such a case the workflow continues to the next step.
4. If the supports() returns "false" then the work flow continues finding whether there is a next element in the handlerAdapter Collection , if found then it moves to step 2. If not then ServletException is thrown, delegating the workflow to phase 6.

**step 4 : Execute handler :**

1. After successfully locating the HandlerAdapter that supports the handler that is located in phase 2, the workflow proceeds to invoke handle() method of HandlerAdapter which further delegates the request to the handler(may be controller).
2. If the handle() method throws any exception then the workflow proceeds to phase 6 , if not it proceeds to phase 5 after collecting the ModelAndView object reference returned by the handle() method. 

**The HandlerAdapter :**

1. The HandlerAdapter implementation takes the responsibility of identifying the type of handler and invokes its appropriate methods.
2. The use of HandlerAdapter facilitates us to use Plain Old Java Objects (POJOs) with any method encapsulating the handler behavior , as a handler.
3. Spring provides the built-in HandlerAdapter implementations supporting different types of handlers to work with.
4. The various types of handlers supported by the Spring built-in HandlerAdapter are controller types, HttpRequestHandler types, Servlet types, and ThrowawayController types.
5. The org.springframework.web.servlet.HandlerAdapter interface declares three methods (supports(), handle(), getLastModified()) that have to be implemented by the HandlerAdapter implementations to help DiapatcherServlet in delegating the request to the handlers.

Spring built-in HandlerAdapter implementations :

1. SimpleControllerHandlerAdapter
2. ThrowawayControllerHandlerAdapter
3. HttpRequestHandlerAdapter
4. SimpleServletHandlerAdapter

Note :

1. By default only the SimpleControllerHandlerAdapter and ThrowawayControllerHandlerAdapter handler adapters are available to the DispatcherServlet.
2. Other handler adapters need to be explicitly configured, as normal as other beans in the context Spring Beans XML configuration file of DispatcherServlet.
3. Even though Spring supports implementing the custom handler adapters that enables handling the request using user defined type of handlers (without making them depend on the Spring API), however it is recommended to use the Spring Controller infrastructure to implement the handler.
4. The custom handler adapter option is given to support some situation where we want to use the existing handlers without rewriting them as spring defined controllers.

The HandlerAdapter uses handler to handle the request and results returning the ModelAndView object.

``` java

package com.spring;

import java.util.Map;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.HandlerAdapter;

import org.springframework.web.servlet.ModelAndView;

public class MyHandlerAdapter implements HandlerAdapter {

public boolean supports(Object handler) {

return (handler instanceof com.spring.MyHandler);

}

public ModelAndView handle(HttpServletRequest request,

HttpServletResponse response, Object handler) throws Exception {

MyHandler myHandler=(MyHandler) handler;

String view=myHandler.process(request,response);

Map model=(Map)request.getAttribute("model");

if(view==null)

return null;

else if(model==null)

return new ModelAndView(view);

else

return new ModelAndView(view,model);

}

public long getLastModified(HttpServletRequest request, Object handler) {

return -1;

}

}

```

``` java

package com.spring;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

public interface MyHandler {

String process(HttpServletRequest request,

HttpServletResponse response) throws Exception;

}

```

##### ModelAndView :

* The ModelAndView is a value object designed to hold model and view making it possible for a handler to return both model and view in a single value.
* The ModelAndView object represents a model and view specified by the handler, which is resolved by the DispatcherServlet using the special framework objects as ViewResolver and View.
* The view is an object that can describe a view name in the form of String which will be resolved by a ViewResolver object to locate View object, alternatively, a View object directly.
* The Model is a Map, enabling to specify multiple objects.

### Phase 5 : Execute Interceptors postHandle methods

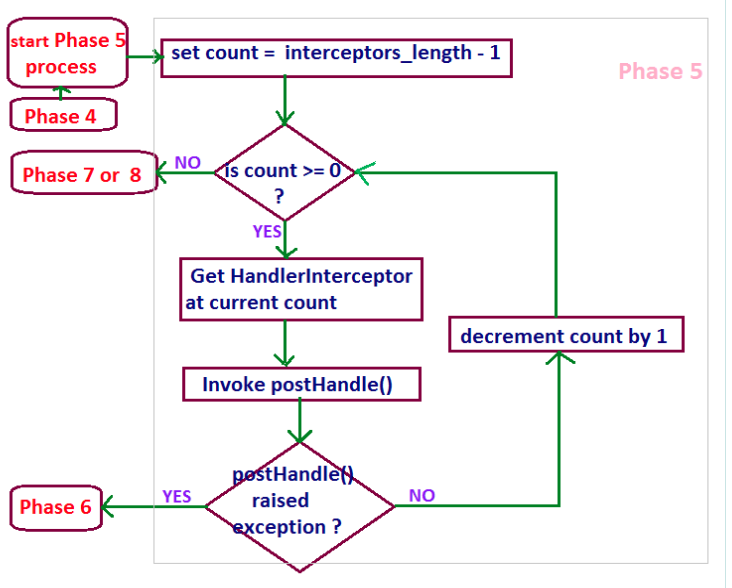
* HandlerInterceptor gives us an opportunity to add common pre- and post-processing behavior without needing to modify each handler implementation.
* The post-processing operations like changing the logical view name in the ModelAndView based on some inputs/output to support different types of views.

**Step 1 : Prepare the Counter :**

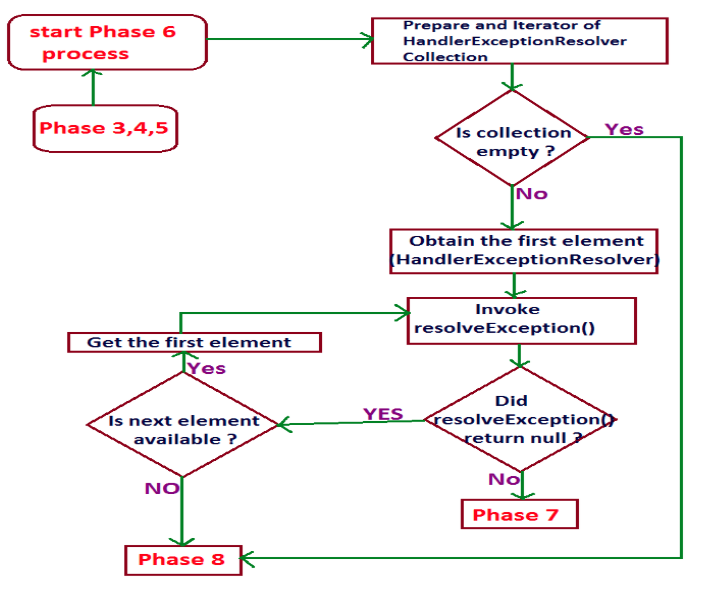
* In this phase workflow starting with setting the count to one minus the interceptor's length so that the postHandle method can be invoked on the interceptor's in the reverse order.
* If the interceptors length is 0(zero) then the workflow proceeds to Phase 7 or 8 based on whether the the handler in Phase 4 has returned a valid ModelAndView object reference or null.
* If there are any interceptors configured then the workflow proceeds to the next step of this phase.

**Step 2 : Invoke postHandle() method :**

* Here it obtains the HandlerInterceptor at the current count and uses it to invoke postHandle() method.
* If the postHandle() method throws any exception then the worlflow proceeds to Phase 6 to handle the exception.
* If the postHandle() method execution is successful (i.e., normal termination) then count is decremented by 1, if the count is grater than or equal to 0(zero) then repeat step2 (i.e., this step) once again.
* If the count is less than 0(zero) then the workflow proceeds to Phase 7 or Phase 8 based on the result of Phase 4.
* That is, if the handler in Phase 4 has returned a valid ModelAndView object reference then workflow proceeds to Phase 7 to render the view, If the handler had returned null then the workflow proceeds to Phase 8 considering that the handler had prepared the response.



### Phase 6 : Handle Exceptions



* This Phase of the Spring Web MVC request processing flow is executed only when there is any exception raised while executing interceptors preHandle() or postHandle() methods or the handler.
* DispatcherServlet uses HandlerExceptionResolver to handle the exceptions thrown by the HandlerInterceptor or handler.

Note : Spring does not take a default type for HandlerExceptionResolver

**Step 1 : Prepare an iterator of exception resolvers :**

* Preparing an iterator of the Collection representing HandlerExceptionResolver objects configured in the application.
* The handlerExceptionResolvers Colletion is initialized in the initialization phase of the DispatcherServlet where it finds all HandlerExceptionResolver's in the ApplicationContext.
* If no HandlerExceptionResolver beans are defined in the application then this collection will be empty.
* If the handlerExceptionResolvers collection is empty then it considers that there are no exceptionhandlers for the application, thus the workflow proceeds to Phase 8 which then ends the request processing by throwing the same exception.
* If the collection is not empty then the workflow continues to the next step of this phase.

**Step 2 : Get HandlerExceptionResolver :**

In this step the work flow obtains the next element from the iterator prepared in step 1, and casts into HandlerExceptionResolver type reference.

**Step 3 : Invoke resolveException() method :**

* After getting the HandlerExceptionResolver of the current iteration the resolveException() method is invoked to handle the exception.
* If the resolveException() method returns a valid ModelAndView object reference then the workflow proceeds to phase 7 for rendering a response.
* If the resolveException() method returns null then it finds whether the next element is available in the iterator.
* If so, then the work flow proceeds to step2 of this phase. If the next element is not available, then the workflow proceeds to phase 8 which then ends the request processing by throwing the same exception.

<beans>

<bean id="handlerExceptionResolver"

class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">

<property name="exceptionMappings">

<props>

<prop key="org.springframework.dao.EmptyResultDataAccessException">

/sqlError.html

</prop>

<prop key="org.springframework.web.HttpRequestMethodNotSupportedException">

/myError.html

</prop>

</props>

</property>

</bean>

</beans>

login.html

<body>

<form action="/login.spring" > <br> // / is optional

User Name : <input type="text" name="userName"> <br>

Password: <input type="password" name="password"> <br>

<input type="submit" value="Login">

</form>

</body>

</html>

LoginController.java

package com.spring;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;

import org.springframework.web.servlet.mvc.Controller;

public class LoginController implements Controller {

LoginModel loginModel;

public void setLoginModel(LoginModel lm){

loginModel=lm;

}

public ModelAndView handleRequest(HttpServletRequest request,

HttpServletResponse response) throws Exception {

String userName=request.getParameter("userName");

String password=request.getParameter("password");

System.out.println("userName : "+userName);

System.out.println("password : "+password);

String type = loginModel.getValidate(userName,password);

System.out.println("type : "+type);

if(type==null)

//return new ModelAndView("/login.html");

throw new MyException("User details are not valid");

else if(type.equals("admin"))

return new ModelAndView("/pages/admin.jsp");

else

return new ModelAndView("/pages/user.jsp");

} //handleRequest

}

LoginModel.java

package com.spring;

import org.springframework.dao.EmptyResultDataAccessException;

import org.springframework.jdbc.core.JdbcTemplate;

public class LoginModel {

private JdbcTemplate jdbcTemplate;

public LoginModel(JdbcTemplate jdbcTemplate) {

this.jdbcTemplate=jdbcTemplate;

}

public String getValidate(String userName, String password) {

String sql="select type from UserDetails where username=\'"

+ userName+ "\' and password=\'"+password+ "\' ";

try{

return jdbcTemplate.queryForObject(sql, String.class);

}catch (EmptyResultDataAccessException e) {

return null;

}

} //getValidate()

}

admin.jsp

Wel come to Admin page <br/>

User Name : <%= request.getParameter("userName") %>

user.jsp

Wel come to User page <br/>

User Name : <%= request.getParameter("userName") %>

myError.html

<body>

A simple test error page <br>

User details are not valid

</body>

ds-servlet.xml

<beans>

<!-- configure DataSource -->

<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">

<property name="driverClassName">

<value>oracle.jdbc.driver.OracleDriver</value>

</property>

<property name="url">

<value>jdbc:oracle:thin:@localhost:1521:xe</value>

</property>

<property name="username">

<value>lms</value>

</property>

<property name="password">

<value>scott</value>

</property>

</bean>

<!-- configure JdbcTemplate -->

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">

<constructor-arg>

<ref local="dataSource" />

</constructor-arg>

</bean>

<bean id="loginModel" class="com.spring.LoginModel">

<constructor-arg>

<ref local="jdbcTemplate"/>

</constructor-arg>

</bean>

<bean id="loginController" class="com.spring.LoginController">

<property name="loginModel">

<ref local="loginModel"/>

</property>

</bean>

<bean id="handlerMapping"

class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">

<property name="mappings">

<props>

<prop key="/login.spring">loginController</prop>

</props>

</property>

</bean>

<bean id="handlerExceptionResolver"

class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">

<property name="exceptionMappings">

<props>

<prop key="com.spring.MyException">

/myError.html

</prop>

</props>

</property>

</bean>

</beans>

web.xml

<web-app>

<servlet>

<servlet-name>ds</servlet-name>

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

<load-on-startup>0</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>ds</servlet-name>

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

</servlet-mapping>

</web-app>

### Phase 7 : Render the View

The Spring Web Mvc request processing workflow executes this phase only in the following two cases :

1. If the handler in Phase 4 retuns a valid(not null) ModelAndView object reference, and phase 5 is executed successfully (i.e., postHandle() method of HandlerInterceptors).
2. If the HandlerExceptionResolver in phase 6 returns a valid (not null) ModelAndView object reference.

**Step 1 : Find the View :**

* The work flow of Phase 7 of Spring Web Mvc request processing work flow starts with finding for the view in the ModelAndView object produced in the previous phases (Phase 4 or 6).
* This is done using the hasView() method of ModelAndView. If hasView() method returns false i.e., the ModelAndView object does not contain a view , then a default view name is set to the ModelAndView object.
* The default view name is obtained by DispatcherServlet using the RequestToViewNameTranslator type object configured in this context.
* If there is no RequestToViewNameTranslator type bean configured in the context then null is set as view name.

**Step 2 : Find the ModelAndView contains View reference :**

* After the successful execution of step 1 the work flow continues with finding whether the view in the ModelAndView object is holding a View object reference or a view name. This is done using the isReference() method of ModelAndView.
* If isReference() method returns 'false' and if the view reference is not null then the work flow proceeds to the next step.
* If isReference() method returns 'true' then the work flow proceeds to resolve the view name. The 'Resolve View Name' process is explained in the next section. If the 'Resolve View Name' process successfully resolves the view name to View object then the workflow proceeds to the next step.
* If the view reference is null or the ViewResolver's fails to resolve the view name(i.e., if 'Resolve View Name' process returns null), the workflow proceeds to phase 8 by throwing a ServletException.

**Step 3 : Delegate to View object for rendering :**

* After successfully locating the View object the request is delegated to the view object invoking render() method of View.
* The view object takes the responsibility to prepare the response, i.e., presentation for the client.
* Thereafter, the workflow proceeds to Phase 8 to perform the finalizations.

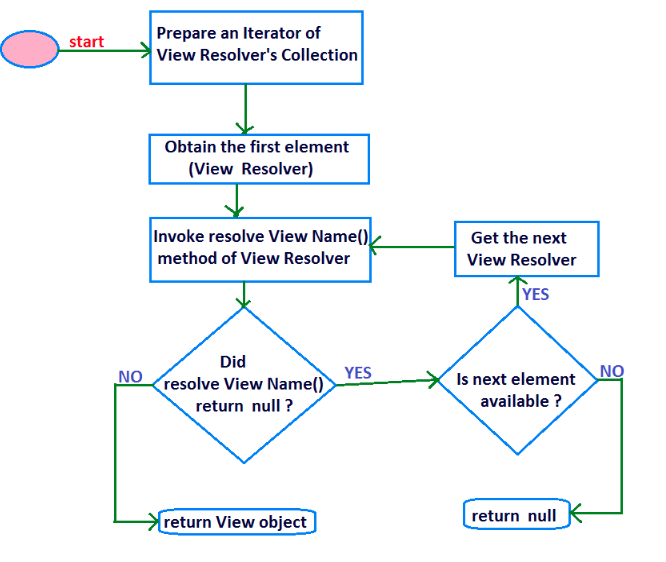
##### The 'Resolve View Name' process :

* As described in the preceding section if the ModelAndView object describes a view name instead of View object, Dispatcherservlet uses the configured ViewResolver object to resolve the view name to locate View object.
* As an overview at this point we can simply consider that the ViewResolver object is responsible to locate the View object that can render a view for this request.

**Step 1 : Prepare a ViewResolver Iterator**

* The wokflow starts with the preparation of an iterator of the collection storing the ViewResolver objects. This collection is created while the DispatcherServlet is being initialized. Thereafter the first element is retrieved from the iterator.

**Step 2 : Resolve View Name**

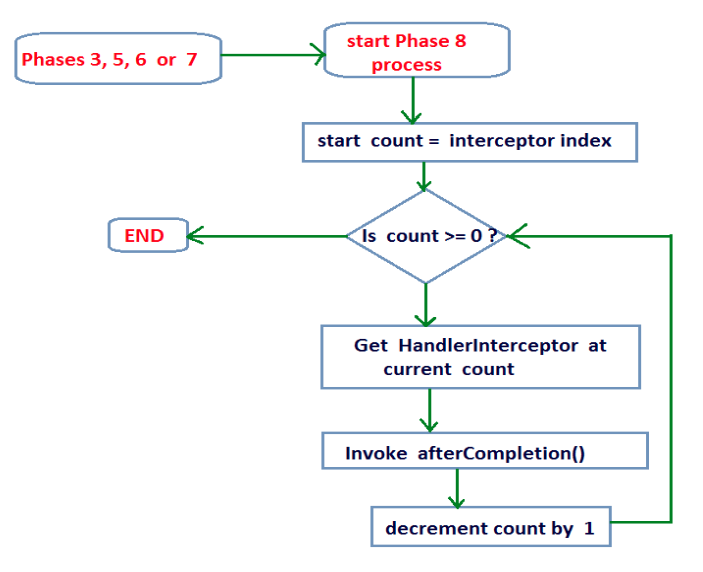
* After step 1 the workflow proceeds to invoke the resolveViewName() method of the current ViewResolver object in the iteration.
* If the resolveViewName() method returns null, which indicates that the current ViewResolver failed to resolve the view name.
* In this case the workflow proceeds to the next step. If the resolveViewName() method returns a valid View object reference then the workflow completes returning the same, which is further used to render the view as described in the preceding section.

**Step 3 : Move to next element**

* If the ViewResolver in the preceding step fails to resolve the view name then the workflow proceeds to get the next ViewResolver from the iterator prepared in step 1.
* If the next element is available then the process moves to step 2, if not the workflow completes returning null.

### Phase 8 : Execute Interceptors afterCompletion methods

* As explained in Phase 3, the HandlerInterceptor gives us an opportunity to add common pre- and post-processing behavior without needing to modify each handler implementation.
* While in Phase 3, the preHandle() method of handler interceptors are executed,
* And in Phase 5, the postHandle() methods are executed.
* In this phase of workflow the afterCompletion() method is executed on the Handler Interceptors whose preHandle() method has successfully executed and returned 'true'(i.e., in Phase 3).
* The afterCompletion() method is called on any outcome of handler execution, thus allowing us to do the resource cleanup.



**Step 1 : Prepare counter :**

* The workflow of Phase 8 starts with setting the count to interceptorIndex value.
* The interceptorIndex is set in the preprocessing phase, i.e., Phase 3 used to track the index of the interceptor in the list that have completed the preHandle processing successfully so that in this phase the afterCompletion methods can be executed only on those HandlerInterceptors.
* That is, if the workflow reaches to this phase from Phase 5 then the interceptorIndex will be equal to the interceptors length minus 1(interceptorsLength - 1) since the preHandle() of all the interceptors are executed successfully.
* Thus the afterCompletion() will be executed on all interceptors in reverse order as compared to the order in which they are configured.
* If the workflow reaches to this phase because of any exception raised in the preHandle() or if preHqndle() returns 'false' then the interceptor chain stops execution the preHandle methods and delegates to this phase via Phase 6 or directly.
* After the counter is prepared if there are any interceptors configured, i.e., if the count is greater than or equal to 0(zero) then the workflow proceeds to step 2, if not it ends the request processing doing the necessary cleanup operations.

**Step 2 : Obtain HandlerInterceptor :**

After the counter is prepared successfully DispatcherServlet obtains the HandlerInterceptor at the current index from the interceptors list.

**Step 3 : Invoke interceptors afterCompletion() method :**

After obtaining the HandlerInterceptor DispatcherServlet invokes afterCompletion() method.

**Step 4 : Change the counter :**

* In the step of Phase 8 workflow the count is decrease by one and then verifies whether the current count is greater than or equal to 0(zero).
* If so then the workflow proceeds to step 2 of this phase, if not it ends the request processing doing the necessary cleanup operations.

With in this phase the Spring Web MVC request processing workflow ends.