**Spring Framework**

Spring is a *lightweight* framework. It can be thought of as a *framework of frameworks* because it provides support to various frameworks such as Struts, Hibernate.

#### **Advantage of Dependency Injection**

* makes the code loosely coupled so easy to maintain
* makes the code easy to test

***Advantages of Spring Framework***

There are many advantages of Spring Framework. They are as follows:

1) Predefined Templates

Spring framework provides templates for JDBC, Hibernate, JPA etc. technologies. So there is no need to write too much code. It hides the basic steps of these technologies.

Let's take the example of JdbcTemplate, you don't need to write the code for exception handling, creating connection, creating statement, committing transaction, closing connection etc. You need to write the code of executing query only. Thus, it save a lot of JDBC code.

2) Loose Coupling

The Spring applications are loosely coupled because of dependency injection.

3) Easy to test

The Dependency Injection makes easier to test the application. The EJB or Struts application require server to run the application but Spring framework doesn't require server.

4) Lightweight

Spring framework is lightweight because of its POJO implementation. The Spring Framework doesn't force the programmer to inherit any class or implement any interface. That is why it is said non-invasive.

5) Fast Development

The Dependency Injection feature of Spring Framework and it support to various frameworks makes the easy development of JavaEE application.

6) Powerful abstraction

It provides powerful abstraction to JavaEE specifications such as JMS, JDBC, JPA and JTA.

**What is different between tight and loose coupling:**

Tight Coupling means one class is dependent on another class.

Loose Coupling means one class is dependent on interface rathar than class.

In Tight coupling, there are hardcoded dependency declared in methods. In Loose coupling, we must pass dependency externally at runtime instead of hardcoded.(Loose couple systems are use interface for decrease dependency with class)

In order to over come tight coupling between objects, spring framework uses dependency injection mechanism with the help of POJO/POJI model and through dependency injection its possible to achieve loose coupling

### ***Inversion Of Control (IOC)***

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly.

IOC makes the code loosely coupled. In such case, there is no need to modify the code if our logic is moved to new environment.

In Spring framework, IOC container is responsible to inject the dependency. We provide metadata to the IOC container either by XML file or annotation.

Container is responsible for creating the bean objects and for injecting it’s dependencies through out our applications.

The main tasks performed by IoC container are:

* to instantiate the application class
* to configure the object
* to assemble the dependencies between the objects

There are two types of IoC containers. They are:

1. **BeanFactory**
2. **ApplicationContext**

The org.springframework.beans.factory.BeanFactory and the org.springframework.context.ApplicationContext interfaces acts as the IoC container. The ApplicationContext interface is built on top of the BeanFactory interface. It adds some extra functionality than BeanFactory such as simple integration with Spring's AOP, message resource handling (for I18N), event propagation, application layer specific context (e.g. WebApplicationContext) for web application. So it is better to use ApplicationContext than BeanFactory.

### ***Using BeanFactory:***

The XmlBeanFactory is the implementation class for the BeanFactory interface. To use the BeanFactory, we need to create the instance of XmlBeanFactory class as given below:

**Resource resource=new ClassPathResource("applicationContext.xml");**

**BeanFactory factory=new XmlBeanFactory(resource);**

The constructor of XmlBeanFactory class receives the Resource object so we need to pass the resource object to create the object of BeanFactory.

### ***Using ApplicationContext:***

The ClassPathXmlApplicationContext class is the implementation class of ApplicationContext interface. We need to instantiate the ClassPathXmlApplicationContext class to use the ApplicationContext as given below:

**ApplicationContext context = new ClassPathXmlApplicationContext("applicationContext.xml");**

The constructor of ClassPathXmlApplicationContext class receives string, so we can pass the name of the xml file to create the instance of ApplicationContext.

#### BeanFactory v/s ApplicationContext in Spring

|  |  |
| --- | --- |
| **BeanFactory (org.springframework.beans)** | **ApplicationContext (org.springframework.context)** |
| Instantiates bean lazily i.e.; when ctx.getBean(“beanName”) is invoked from application Eg; BeanFactory bf = new XmlBeanFactory(“SpringXml”); | Instantiates bean eagerly i.e.; upon ApplicationContext startup/loaded. eg; ApplicationContext app = new ClassPathXmlApplicationContext(“SpringXml”); |
| No Supports for I18N. | Supports for Internalization (I18N) |
| No Supports for Annotation | Supports Annotation based Dependency Injection (DI) |
| Recommended to use in small lightweight application like Mobile, Applet, etc | AppplicationContext best suited for enterprise application |
| Very limited access i.e.; low level resources | It is very convenient to load resources from varios resources like ClassPath, FileSystem, etc. |
| No such support | Provides generic way to load resources such as Image file |
| Loading multiple configuration files is not possible | In large enterprise project, you have to load multiple configuration files. In that case, ApplicationContext is very good option Eg.; new ClassPathXmlApplicationContext(new String[]{“confg-1”, “confg-2”, “confg-3”}); |

**AutoWire:**

Wiring a bean means configuring a bean along with its dependencies into an xml file like previous concepts, by default autowiring is disabled in spring framework. It means the programmer has to explicitly wire the bean properties into an xml file.

If autowiring is enabled then spring container will take care about injecting the dependencies, programmer no need to configure into an xml file explicitly. Autowiring is only supported if the dependancies are in the form of objects only

**autowire byname:**

For this type of autowiring, setter method is used for dependency injection. Also the

in this case, spring framework attempts to find out a bean in the configuration file, whose id is matching with the property name to be wired.  If a bean found with id as property name then that class object will be injected into that property by calling setter injection.  If no id is found then that property remains un-wired, but never throws any exception.

**autowire byType**: For this type of autowiring, class type is used. So there should be only one bean configured for this type in the spring bean configuration file, it is used setter injection

**Autowire by constructor:**  This is almost similar to autowire byType, the only difference is that constructor is used to inject the dependency.

**Autodetect:**

Spring first tries to wire using autowire by *constructor*, if it does not work, Spring tries to autowire by *byType*

**Annotations:**

**@Autowired annotation:**

This annotation is applied on fields, setter methods, and constructors, The @Autowired annotation injects object dependency implicitly.

**@Qualifier annotation**: This annotation is used to avoid conflicts in bean mapping and we need to provide the bean name that will be used for autowiring. This way we can avoid issues where multiple beans are defined for same type. This annotation usually works with the @Autowired annotation. For constructors with multiple arguments, we can use this annotation with the argument names in the method.

By default spring bean autowiring is turned off. Spring bean autowire default value is “default” that means no autowiring is to be performed. autowire value “no” also have the same behavior.

**1. @Resource – Defined in the javax.annotation package and part of Java:**

@Resource is quite similar to @Autowired and @Inject, but the main difference is the execution paths taken to find out the required bean to inject. @Resource will narrow down the search first by name then by type and finally by Qualifiers (ignored if match is found by name). @Autowired and @Inject will narrow down the search first by type then by qualifier and finally by the name.

Standard @Resource annotation marks a resource that is needed by the application. It is analogous to @Autowired in that both injects beans by type when no attribute provided. But with name attribute, @Resource allows you to inject a bean by it’s name, which @Autowired does not.

    @Resource(name="applicationUser")

    private ApplicationUser user;

In above code, Application’s user property is annotated with @Resource(name=”applicationUser”). In this case, a bean with name ‘applicationUser’ found in applicationContext will be injected here.

@Resource annotation does not have this flexibility. In case the dependency annotated with @Resource not found, Spring will throw an exception

**2 @Inject – Defined in the javax.inject package and part of Java:**

This annotation is an almost complete drop-in replacement for Spring’s @Autowired annotation. So, instead of using the Spring-specific @Autowired annotation, you might choose to use @Inject. One of the differences between @Autowired and @Inject is that @Inject does not have the required field so in case we fail to find a suitable object to injected it will fail while

**3 @Autowired – Defined in the package org.springframework.bean.factory and part of Spring framework.**

#### **Mark Autowiring optional with attribute required=”false”:**

@Autowired required attribute to ‘false’ will make this filed optional for autowiring and Spring will skip it(remain null) if dependency not found.

@Autowired(required=false)

private License license;

if no bean of type License been found, it will remain null and no error will be thrown on context loading.

**@Configuration:**

Aannotated class are same like old XML file where we used to define beans. @Configuration is configuring the beans using, @Configuration will have methods to instantiated and configure the dependencies. Such methods will be annotated with **@Bean.**

**@ComponentScan** which will make Spring auto detect the annotated beans via scanning the specified package and wire them wherever needed (using @Resource or @Autowired ).

Example:

@Configuration

@ComponentScan("com.websystique.spring")

public class AppConfig {

}

analogs

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="<http://www.springframework.org/schema/beans>"

    xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>" xmlns:context="<http://www.springframework.org/schema/context>"

    xsi:schemaLocation="<http://www.springframework.org/schema/beans> <http://www.springframework.org/schema/beans/spring-beans-4.0.xsd>

<http://www.springframework.org/schema/context> <http://www.springframework.org/schema/context/spring-context-4.0.xsd>">

    <context:component-scan base-package="com.websystique.spring" />

</beans>

**@**Bean**:**

This annotation is used at the method level. @Bean annotation works with @Configuration to create Spring beans. As mentioned earlier, @Configuration will have methods to instantiate and configure dependencies.

It is a method level annotation which work with @Configuration as mentioned above. It is same like <bean> element in XML configuration. It can also be used in **Component Class(**Annotated with @Configuration**)**.

@Configuration

public class AppConfig {

@Bean

public ContactService contactService() {

return new ContactServiceImpl();

}

Above configuration is same as

<beans>

<bean name="contactService" class="com.ContactServiceImpl"/>

</beans>

**Properties file Reader in spring:**

XmlType configuration:

<context:property-placeholder location="classpath:db.properties,app.properties" />

<context:property-placeholder location="classpath:app11.properties" ignore-resource-not-found="true" />

**Java Configuration:**

@Configuration

@PropertySource(value="classpath:missing.properties", ignoreResourceNotFound=true)

public class AppConfig {}

@PropertySources({

@PropertySource(value = "classpath:missing.properties", ignoreResourceNotFound=true),

@PropertySource("classpath:config.properties")

})

Reading Value:

@Value("${db.host.url:nage}")

private String dbHost;

**@ConfigurationProperties:**

Spring Boot @ConfigurationProperties is letting developer maps the entire file into an object easily.

@Value annotation to inject the .properties value one by one, this is good for small and simple structure .properties files

@Component

@ConfigurationProperties("app") // prefix app, find app.\* values

public class AppProperties {

}

**Scopes:**

**singleton (default):** Single bean object instance per spring IoC container

**prototype**: Opposite to singleton, it produces a new instance each and every time a bean is requested.

**Request :** A single instance will be created and available during complete lifecycle of an HTTP request.

Only valid in web-aware Spring ApplicationContext.

**request:** With this bean scope, a new bean instance will be created for each web request made by client. As soon as request completes, bean will be out of scope and garbage collected.

**session:** Just like request scope, this ensures one instance of bean per user session. As soon as user ends its session, bean is out of scope.

**singleton scope:**

singleton is default bean scope in spring container. It tells the container to create and manage only one instance of bean class, per container. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached instance.

Example of singleton scope bean using Java config –

@Component

//This statement is redundant - singleton is default scope

@Scope("singleton") //This statement is redundant

public class BeanClass {

}

Example of singleton scope bean using XML config –

<!-- To specify singleton scope is redundant -->

<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="singleton" />

//or

<bean id="beanId" class="com.howtodoinjava.BeanClass" />

**2 prototype scope**:

prototype scope results in the creation of a new bean instance every time a request for the bean is made by application code.

You should know that destruction bean lifecycle methods are not called prototype scoped beans, only initialization callback methods are called. So as developer, you are responsible for clean up prototype-scoped bean instances and any resource there hold.

Java config example of prototype bean scope –

@Component

@Scope("prototype")

public class BeanClass {

}

XML config example of prototype bean scope –

<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="singleton" />

**As a rule, you should prefer to use the prototype scope for all stateful beans and the singleton scope for stateless beans.**

To use beans in the request, session, application and websocket scopes, you need to register the RequestContextListener or RequestContextFilter.

**3 request scope:**

In request scope, container creates a new instance for each and every HTTP request. So, if server is currently handling 50 requests, then container can have at most 50 individual instances of bean class. Any state change to one instance, will not be visible to other instances. These instances are destructed as soon as the request is completed.

Java config example of request bean scope –

@Component

@Scope("request")

public class BeanClass {

}

//or

@Component

@RequestScope

public class BeanClass {

}

XML config example of request bean scope –

<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="request" />

Read More: How web servers work?

**4 session scope:**

In session scope, container creates a new instance for each and every HTTP session. So, if server has 20 active sessions, then container can have at most 20 individual instances of bean class. All HTTP requests within single session lifetime will have access to same single bean instance in that session scope.

Any state change to one instance, will not be visible to other instances. These instances are destructed as soon as the session is destroyed/end on server.

Java config example of session bean scope –

@Component

@Scope("session")

public class BeanClass {

}

//or

@Component

@SessionScope

public class BeanClass {

}

XML config example of session bean scope –

<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="session" />

**5 application scope:**

In application scope, container creates one instance per web application runtime. It is almost similar to singleton scope, with only two differences i.e.

application scoped bean is singleton per ServletContext, whereas singleton scoped bean is singleton per ApplicationContext. Please note that there can be multiple application contexts for single application.

application scoped bean is visible as a ServletContext attribute.

Java config example of application bean scope –

@Component

@Scope("application")

public class BeanClass {

}

//or

@Component

@ApplicationScope

public class BeanClass {

}

XML config example of application bean scope –

<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="application" />

**6 websocket scope:**

The WebSocket Protocol enables two-way communication between a client and a remote host that has opted-in to communication with client. WebSocket Protocol provides a single TCP connection for traffic in both directions. This is specially useful for multi-user applications with simultaneous editing and multi-user games.

In this type of web applications, HTTP is used only for the initial handshake. Server can respond with HTTP status 101 (switching protocols) if it agrees – to handshake request. If the handshake succeeds, the TCP socket remains open and both client and server can use it to send messages to each other.

Java config example of websocket bean scope –

@Component

@Scope("websocket")

public class BeanClass {

}

XML config example of websocket bean scope –

<bean id="beanId" class="com.howtodoinjava.BeanClass" scope="websocket" />

LaZy Initialization:

By default, Spring “application context” eagerly creates and initializes all ‘[singleton scoped](https://howtodoinjava.com/design-patterns/creational/singleton-design-pattern-in-java/)‘ beans during application startup itself. It helps in detecting the bean configuration issues at early stage, in most of the cases. But sometimes, you may need to mark some or all beans to be lazy initialized due to different project requirements.

Xml:

<beans default-lazy-init="true">

 <bean id="employeeManager" class="com.howtodoinjava.spring.service.impl.EmployeeManagerImpl" />

 <beans>

Java

@Configuration

public class AppConfig {

    @Lazy

    @Bean

    public EmployeeManager employeeManager() {

        return new EmployeeManagerImpl();}

}

Load all beans lazy

@Lazy

@Configuration

public class AppConfig {

    @Bean

    public EmployeeManager employeeManager() {

        return new EmployeeManagerImpl();

    }

}

**Spring MVC**

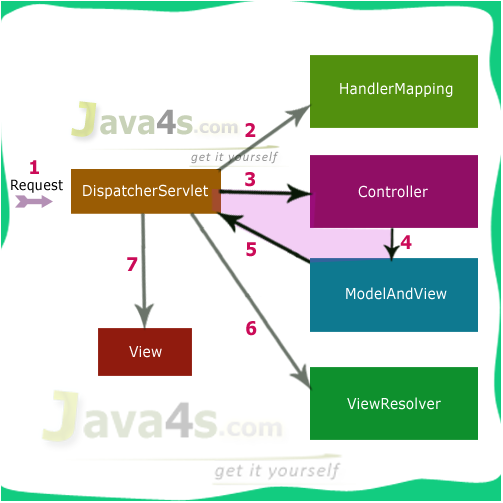
**MVC:**

**The Model** can be some DAO layer or some Service Layers which give some information about request or requested information or Model can be a POJO which encapsulates the application data given by the controller.

**The View** is responsible for rendering the model data and in general it generates HTML output that the client’s browser can interpret.

**The Controller** is responsible for processing user requests and building appropriate model and passes it to the view for rendering.

DispatcherServlet acts as [front controller](http://www.baeldung.com/java-front-controller-pattern) – receiving all incoming HTTP requests and processing them. It is inherited from [javax.servlet.http.HttpServlet](https://tomcat.apache.org/tomcat-7.0-doc/servletapi/javax/servlet/http/HttpServlet.html), it is typically configured in the web.xml file.



Step 1: First request will be received by DispatcherServlet

Step 2: DispatcherServlet will take the help of HandlerMapping and get to know the Controller class name associated with the given request

Step 3: So request transfer to the Controller, and then controller will process the request by executing appropriate methods and returns ModeAndView object (contains Model data and View name) back to the DispatcherServlet

Step 4: Now DispatcherServlet send the model object to the ViewResolver to get the actual view page

Step 5: Finally DispatcherServlet will pass the Model object to the View page to display the result

**HandlerMappings:**

HandlerMapping is an interface that defines a mapping between requests and [handler objects](http://www.baeldung.com/spring-mvc-handler-adapters), While Spring MVC framework provides some ready-made implementations, by default **DispatcherServlet** uses **BeanNameUrlHandlerMapping** and **DefaultAnnotationHandlerMapping**

[Spring MVC](https://www.javainterviewpoint.com/category/spring-mvc/) we majorly use the below handler mappings.

1. BeanNameUrlHandlerMapping
2. [ControllerClassNameHandlerMapping](http://www.javainterviewpoint.com/spring-mvc-controllerclassnamehandlermapping-example/)
3. [SimpleUrlHandlerMapping](http://www.javainterviewpoint.com/spring-mvc-simpleurlhandlermapping-example/)

***BeanNameUrlHandlerMapping*** maps request URLs to beans with the same name.

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

<bean name="/helloWorld.htm" class="com.javainterviewpoint.HelloWorldController" />

<bean name="/hello\*" class="com.javainterviewpoint.HelloWorldController" />

<bean name="/welcome.htm" class="com.javainterviewpoint.WelcomeController"/>

helloWorld.htm is requested, the DispatcherServlet redirects it to the HelloWorldController.

hello123 is requested, the DispatcherServlet redirects it to the HelloWorldController.

welcome.htm is requested, the DispatcherServlet redirects it to the WelcomeController.

welcome123 is requested, you will get 404 error as there is no mapping for it.

@Configuration

public class BeanNameUrlHandlerMappingConfig {

@Bean

BeanNameUrlHandlerMapping beanNameUrlHandlerMapping() {

return new BeanNameUrlHandlerMapping();

}

@Bean("/beanNameUrl")

public WelcomeController welcome() {

return new WelcomeController();

}

}

***ControllerClassNameHandlerMapping*** maps URL to a registered controller bean (or a controller annotated with the *@Controller* annotation) that has, or starts with, the same name

remove the *“Controller”* suffix, then change the name to a lower case and return it as the mapping with a leading *“/”*.

For example *“WelcomeController”* would return as mapping to *“/welcome\*”*, i.e. to any URL that starts with *“welcome”*.

bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping" />

<bean class="com.baeldung.WelcomeController" /

@Configuration

public class ControllerClassNameHandlerMappingConfig {

@Bean

public ControllerClassNameHandlerMapping controllerClassNameHandlerMapping() {

return new ControllerClassNameHandlerMapping();

}

@Bean

public WelcomeController welcome() {

return new WelcomeController(); } }

When using **ControllerClassNameHandlerMapping**, there is no need for  **bean name**

***SimpleUrlHandlerMapping****is the most flexible handler mapping class,helps to explicitly-map URLs with their controllers respectively.*

*<bean class = "org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">*

*<property name = "mappings">*

*<props>*

*<prop key = "/welcome.htm">welcomeController</prop> <prop key = "/helloWorld.htm">helloController</prop>*

*</props>*

*</property>*

*</bean>*

*<bean id = "helloController" class = "com.tutorialspoint.HelloController" />*

*<bean id = "welcomeController" class = "com.tutorialspoint.WelcomeController"/>*

*@Configuration*

*public class SimpleUrlHandlerMappingConfig {*

*@Bean*

*public SimpleUrlHandlerMapping simpleUrlHandlerMapping() {*

*SimpleUrlHandlerMapping simpleUrlHandlerMapping*

*= new SimpleUrlHandlerMapping();*

*Map<String, Object> urlMap = new HashMap<>();*

*urlMap.put("/simpleUrlWelcome", welcome());*

*simpleUrlHandlerMapping.setUrlMap(urlMap);*

*return simpleUrlHandlerMapping;*

*}*

*@Bean*

*public WelcomeController welcome() {*

*return new WelcomeController();*

*}}*

InternalResourceViewResolver in Spring MVC is used to mapping the logical view names to actual view files under a particular directory of the web application. This logical view name is returned by the handler method of controller. In Spring MVC InternalResourceViewResolver is used to resolve “internal resource view” (i.e. JSP under the WEB-INF directory) based on a predefined URL pattern.

How to Resolve View

In Spring MVC, InternalResourceViewResolver allow us to add some predefined prefix and suffix to the view name (prefix + view name + suffix), and generate the final view page URL as below.

**XML Configuration:**

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">

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

<property name="prefix" value="/WEB-INF/view/"/>

</bean>

In Java Configuration:

@Bean

public ViewResolver viewResolver(){

InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();

viewResolver.setPrefix("/WEB-INF/view/");

viewResolver.setSuffix(".jsp");

return viewResolver;

}

*Spring Mvc Application flow Examples:*

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://java.sun.com/xml/ns/javaee" xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app\_3\_0.xsd" id="WebApp\_ID" version="3.0">  <display-name>spring-mvc-example</display-name>  <!-- Add Spring MVC DispatcherServlet as front controller -->  <servlet>  <servlet-name>spring</servlet-name>  <servlet-class>  org.springframework.web.servlet.DispatcherServlet  </servlet-class>  <init-param>  <param-name>contextConfigLocation</param-name>  <param-value>/WEB-INF/spring-servlet.xml</param-value>  </init-param>  <load-on-startup>1</load-on-startup>  </servlet>  <servlet-mapping>  <servlet-name>spring</servlet-name>  <url-pattern>/</url-pattern>  </servlet-mapping>  </web-app>  contextConfigLocation init-param is used to provide the location of spring bean configuration file.    One thing to note here is the name of servlet in <servlet-name> tag in web.xml. Once the DispatcherServlet is initialized, it will looks for a file name [servlet-name]-servlet.xml in WEB-INF folder of web application. In this example, the framework will look for file called spring-servlet.xml.  <?xml version="1.0" encoding="UTF-8"?>  <beans>  <!-- DispatcherServlet Context: defines this servlet's request-processing  infrastructure -->  <!-- Enables the Spring MVC @Controller programming model -->  <annotation-driven />  <context:component-scan base-package="com.journaldev.spring" />  <!-- Resolves views selected for rendering by @Controllers to .jsp resources in the /WEB-INF/views directory -->  <beans:bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">  <beans:property name="prefix" value="/WEB-INF/views/" />  <beans:property name="suffix" value=".jsp" />  </beans:bean>  </beans:beans>  There are three important configurations.   * annotation-driven tells DispatcherServlet to look for Controller classes using @Controller annotation. * context:component-scan tells DispatcherServlet where to look for controller classes. * InternalResourceViewResolver bean configuration to specify location of view pages and suffix used. Controller class methods return name of the view page and then suffix is added to figure out the view page to use for rendering the response. |

Spring xml files laading using spring context file:

<import resource="common/Spring-Common.xml"/>

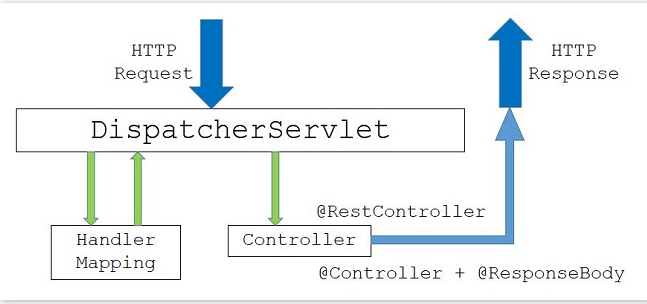
@Import({JmsJndiTemplateConfig.class, JmsConnectionFactoryConfig.class})

<!-- Enables the Spring MVC @Controller programming model -->

<annotation-driven />

**RESTful Web Service Controller and Difference between @Controller – @RestControler:**

The main difference from RESTful Web Service Controller is the way the HTTP response body is created. RESTful Controller returns the object which data is written directly to HTTP response under JSON or XML without using View Resolver.



With new approach, we can return data directly from Controller without finding appropriate View by using annotation @ResponseBody over method.

Hence:

**@Controller:**  is used to mark classes as Spring MVC Controller

**@RestController:** Is the same as @Controller, but added @ResponseBody annotation (that indicates a method return value should be bound to the web response body.

**@ResponseBody** :

If any handler method of controller class is annotated with @ResponseBody, Spring converts the return value of this method and writes it to the http response automatically.

if a method is annotated with @ResponseBody, Spring will bind the return value to outgoing HTTP response body. While doing that, Spring will [behind the scenes] use HTTP Message converters to convert the return value to HTTP response body [serialize the object to response body], based on Content-Type present in request HTTP header. As already mentioned, in Spring 4, you may stop using this annotation.

What is Behind the @ResponseBody

There is a list of HttpMessageConverters registered in the background when we enable Spring web mvc feature to the web application by using @EnableWebMvc or <mvc:annotation-driven> namespace. HTTPMessageConverter is responsible to convert the request body to a specific class and back to the response body again, depending on a predefined mime type.

**@RequestParam** is use for query parameter (static values) <http://localhost:8080/calculation/pow?base=2&ext=4>

**@PathVariable** is use for dynamic values.

like: <http://localhost:8080/calculation/sqrt/8>

|  |
| --- |
| @RequestMapping(value="/pow", method=RequestMethod.GET)  public int pow(@RequestParam(value="base") int base1, @RequestParam(value="ext") int ext1){  int pow = (int) Math.pow(base1, ext1);  return pow;  }  @RequestMapping("/sqrt/{num}")  public double sqrt(@PathVariable(value="num") int num1){  double sqrtnum=Math.sqrt(num1);  return sqrtnum;  } |

**ModelAndView:**

ModelAndView is an object that holds both the model and view. The handler returns the ModelAndView object and DispatcherServlet resolves the view using View Resolvers and View.

ModelAndView model = **new** ModelAndView("employeeDetails");

model.addObject("employeeObj", **new** EmployeeBean(**123**));

model.addObject("msg", "Employee information.");

**return** model;

*collection list display:*

<c:if test="${not empty lists}">

<ul>

<c:forEach var="listValue" items="${lists}">

<li>${listValue}</li>

</c:forEach>

</ul>

</c:if>

If you are using XML based configuration then use <mvc:annotation-driven/> as an alternative to @EnableWebMvc

**Java Based mvc configuration without xml file:**

AbstractAnnotationConfigDispatcherServletInitializer Example

This class extends AbstractDispatcherServletInitializer and does few things implicitly which otherwise you could be doing yourself. An added advantage is that you now can use the convenience classes provided by Spring instead of manually configuring the DispatcherServlet and/or ContextLoaderListener.

It is the preferred approach for applications that use Java-based Spring configuration. It enables you to start servlet application context as well as root application context.

|  |
| --- |
| public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {@Overrideprotected Class<?>[] getRootConfigClasses() {return new Class[] { RootConfig.class };}@Overrideprotected Class<?>[] getServletConfigClasses() {return new Class[] { WebMvcConfig.class };}@Overrideprotected String[] getServletMappings() {return new String[] { "/" };}} @Configuration  @EnableWebMvc  @ComponentScan(basePackages = { "com.howtodoinjava.demo.spring"})  public class WebMvcConfig implements WebMvcConfigurer {    @Bean  public InternalResourceViewResolver resolver() {  InternalResourceViewResolver resolver = new InternalResourceViewResolver();  resolver.setViewClass(JstlView.class);  resolver.setPrefix("/WEB-INF/views/");  resolver.setSuffix(".jsp");  return resolver;  }  } |

#### **Spring Cache:**

The caching feature can be declaratively enabled by simply adding the @EnableCaching annotation to any of the configuration classes:

In xml: <cache:annotation-driven />

**Use Caching With Annotations:**

After enabling the cache we can use following list of declarative annotations.

@Cacheable

@CacheEvict

@CachePut

@Caching

@CacheConfig

**@Cacheable:** It is one of the most important and common annotation for caching the requests. If you annotate a method with @Cacheable, if multiple requests are received by the application, then this annotation will not execute the method multiple times, instead it will send the result from the cached storage.

The simplest way to enable caching behavior for a method is to demarcate it with @Cacheable and parameterize it with the name of the cache where the results would be stored:

@Cacheable(value="cities")

public List<City> findAllCity(){

return (List<City>) cityRepository.findAll();

}

The findAllCity() call will first checks the cache cities before actually invoking the method and then caching the result.

Spring framework also supports multiple caches to be passed as parameters:

@Cacheable(value={"cities","city-list"})

public List<City> findAllCity(){

return (List<City>) cityRepository.findAll();

}

**@CacheEvict:**

If we annotate all methods with @Cacheable then the size of cache may be some problem. We don’t want to populate the cache with values that we don’t need often. Caches can grow quite large, quite fast, and we could be holding on to a lot of stale or unused data. @CacheEvict annotation is used for removing a single cache or clearing the entire cache from the cache storage so that fresh values can be loaded into the cache again:

@CacheEvict(value="cities", allEntries=true)

public List<City> findAllCity(){

return (List<City>) cityRepository.findAll();

}

Here allEntries indicated whether all the data in the cache has to be removed.

**@CachePut:**

@CachePut annotation helps for updating the cache with the latest execution without stopping the method execution. The difference between @Cacheable and @CachePut is that @Cacheable will skip running the method, whereas @CachePut will actually run the method and then put its results in the cache.

@CachePut(value="cities")

public List<City> findAllCity(){

return (List<City>) cityRepository.findAll();

}

**@Caching:**

What if you want to use multiple annotations of the same type for caching a method? @Caching annotation used for grouping multiple annotations of the same type together when one annotation is not sufficient for the specifying the suitable condition. For example, you can put multiple @CacheEvict or @CachePut annotation inside @Caching to narrow down your conditions as you need.

@Caching(evict = {

@CacheEvict("cities"),

@CacheEvict(value="city-list", key="#city.name") })

public List<City> findAllCity(){

return (List<City>) cityRepository.findAll();

}

**@CacheConfig:**

You can annotate @CacheConfig at the class level to avoid repeated mentioning in each method. For example, in the class level you can provide the cache name and in the method you just annotate with @Cacheable annotation.

@CacheConfig(cacheNames={"cities"})

public class CityMasterService {

@Cacheable

public List<City>) findAllCity() {

return (List<City>)) cityRepository.findAll();

}

}

**Conditional Caching:**

Sometimes we don’t want to cache some result there are no need to cache. For example – reusing our example from the @CachePut annotation – this will both execute the method as well as cache the results each and every time:

@CachePut(value="cities")

public List<City>) findAllCity(State state){

return (List<City>)) cityRepository.findAll(state.getStateCode());

}

**Condition Parameter:**

@CachePut(value="cities", condition="#state.stateName=='UP'")

public ListList<City>) findAllCity(State state){

return (List<City>)) cityRepository.findAll(state.getStateCode());

}

@CachePut can be parameterized with a condition parameter that takes a SpEL expression to ensure that the results are cached based on evaluating that expression.

Unless Parameter:

We can also control the caching based on the output of the method rather than the input – via the unless parameter

@CachePut(value="cities", unless="#result.length()>25")

public List<City>) findAllCity(String state){

return (List<City>)) cityRepository.findAll(state.getStateCode());

}

|  |
| --- |
| Cache.xml file:  <ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="true"  monitoring="autodetect" dynamicConfig="true">  <!-- By default, Ehcache stored the cached files in temp folder. -->  <!-- <diskStore path="java.io.tmpdir" /> -->    <!-- Ask Ehcache to store cache in this path -->  <diskStore path="c:\\cache" />  <!-- Sample cache named cache1  This cache contains a maximum in memory of 10000 elements, and will expire  an element if it is idle for more than 5 minutes and lives for more than  10 minutes.  If there are more than 10000 elements it will overflow to the  disk cache, which in this configuration will go to wherever java.io.tmp is  defined on your system. On a standard Linux system this will be /tmp" -->  <cache name="cache1"  maxEntriesLocalHeap="10000"  maxEntriesLocalDisk="1000"  eternal="false"  diskSpoolBufferSizeMB="20"  timeToIdleSeconds="300" timeToLiveSeconds="600"  memoryStoreEvictionPolicy="LFU"  transactionalMode="off">  <persistence strategy="localTempSwap" />  </cache>  </ehcache>  @Configuration  @EnableCaching  @ComponentScan({ "com.mng.spring.SpringBootCache" })  **public** **class** AppConfig {  **public** CacheManager CacheManager(){  **return** **new** EhCacheCacheManager(ehCacheCacheManager().getObject());  }    @Bean  **public** EhCacheManagerFactoryBean ehCacheCacheManager(){  EhCacheManagerFactoryBean cmfb = **new** EhCacheManagerFactoryBean();  cmfb.setConfigLocation(**new** ClassPathResource("ehcache.xml"));  cmfb.setShared(**true**);  **return** cmfb;  }} |

#### org.springframework.cache.CacheManager is the common Cache abstraction provided by spring to handle all caching related activities. CacheManager controls and manages Caches [ org.springframework.cache.Cache] and can be used to retrieve these for storage

### 1. @Cacheable

Used for Cache-population. @Cacheable annotation indicates that the result of invoking a method (or all methods in a class) can be cached. Almost anything [object,array,list,..] can be cached. A cache itself can be imagined as a key-value based store. First time a method annotated with @Cacheable gets called, it gets executed and it’s return value is stored in Cache using a key[method parameter for instance, ]. Next time, if the method gets called using same key[same parameter for instance], the result is returned directly from Cache, **without executing the method**.

@Cacheable annotation supports many optional attributes to control cache population. These attributes can use SpEL to specify the caching criteria.

value : Specifies the name of the cache being used.  
key : Used to specify the key for cache storage. You can use SpEL to specify the key.

|  |
| --- |
| @Cacheable(value="products", key="#product.name")  public Product findProduct(Product product){//product name will be used as a key  ..  return aproduct;  } |

**Important:** If you missed to provide the ‘key’ attribute, Spring may generate the key based on method argument itself [product as a key]. Hence you must make sure to implement **hashCode() and equals()** for that modal object. In contrast, you can use [KeyGenerator](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/cache/interceptor/KeyGenerator.html) to generate a key for you.

condition : Conditional Caching. Item will be cached only if the condition mentioned in ‘condition’ met. Note that condition applies to method argument and evaluated before method execution.

|  |
| --- |
| @Cacheable(value="products", key="#product.name", condition="#product.price<500")  public Product findProduct(Product product){  ..  return aproduct;  } |

unless : Conditional Caching, applies to return value of method. Item will be cached, unless the condition mentioned in ‘unless’ met. Note that condition applies to return value of method.#result refers to method return value.

|  |
| --- |
| @Cacheable(value="products", key="#product.name", condition="#product.price<500", unless="#result.outofstock")  public Product findProduct(Product product){  ..  return aproduct;  } |

Multiple Caches:  
@Cacheable can use multiple caches at the same time. In this situation, a requested item will be checked in all the mentioned cached and if it found in any of them, method will not be executed.If it does not exist in any of the cache, method will gets executed and it’s result will be stored in all of those caches.

|  |
| --- |
| @Cacheable({"products", "items"})  public Product findProduct(Product product) {...  ..  return aproduct;  } |

### @CachePut

Used for Cache-update operation. Method annotated with @CachePut are **always gets executed** and there result gets stored in the cache, eventually overriding any entry with same key in cache. @CachePut, like @Cacheable, supports several attributes, having similar functionality as described above.

Think about a product-refresh operation, where we want a specific product to be re-calculated [may be due to a new price] and then store that product in cache for any future reference. Note that while @CacheEvict is used to remove an item[or all of them] from cache, @CachePut is to update an item.

|  |
| --- |
| @CachePut(value = "products", key = "#product.name" , unless="#result==null")  public Product updateProduct(Product product) {      logger.info("<!----------Entering updateProduct ------------------->");      for(Product p : products){          if(p.getName().equalsIgnoreCase(product.getName()))              p.setPrice(product.getPrice());              return p;      }      return null;  } |

#### @CacheEvict

Used for Cache-removal /cache-cleanup operation. @CacheEvict annotation indicates that a method (or all methods on a class) triggers a cache evict operation, removing specific [or all] items from cache. Various attributes provides complete control to enforce the required behavior for cache-eviction.

|  |
| --- |
| @CacheEvict(value = "products", key = "#product.name")  public void refreshProduct(Product product) {      //This method will remove only this specific product from 'products' cache.  }    @CacheEvict(value = "products", allEntries = true)  public void refreshAllProducts() {      //This method will remove all 'products' from cache, say as a result of flush-all API.  } |

### @Caching

@Caching annotation comes handy when you want to specify multiple annotations of the same type, such as @CacheEvict or @CachePut on same method.

Let’s say you have two caches containing same product using same keys. Now, if you want to evict the specific product from both caches, it’s straight forward.

Let’s say you have two caches containing same product using same keys. Now, if you want to evict the specific product from both caches, it’s straight forward.

|  |
| --- |
| @CacheEvict(value = {"products", "items"}, key = "#product.name")  public void refreshProduct(Product product) {      //This method will remove only this specific product from 'products' & 'items' cache.  } |

But what if they are using different keys? You may think something like below would be good enough.

|  |
| --- |
| @CacheEvict(value = "products", key = "#product.name")  @CacheEvict(value = "items"  ,  key = "#product.id")  public void refreshProduct(Product product) {      //This method will remove only this specific product from 'products' & 'items' cache.  } |

Instead you will get a compiler error, as it is not allowed by the language itself to have two annotations of the same time on same element.

@Caching to the rescue.

|  |
| --- |
| @Caching(evict = {          @CacheEvict(value = "products", key="#product.name"),          @CacheEvict(value = "items"   , key = "#product.id")  })  public void refreshProduct(Product product) {      //This method will remove only this specific product from 'products' & 'items' cache.  } |

### @CacheConfig

@CacheConfig is a class-level annotation which can be used to specify the common caching related settings directly on class level, thus freeing user from duplicating them on each method level. You can of course override the setting specified on class level, on individual method. Common configuration setting that can be specified at class level are cache names, custom KeyGenerator, the custom CacheManager & custom CacheResolve.

|  |
| --- |
| @CacheConfig(value="products", keyGenerator="myKeyGenerator")  class MyClass{        @Cacheable      public Product findProduct(Product product) {...          ..          return aproduct;      }        @Cacheable(value="items")      public Product findSoldProduct(Product product) {...          ..          return aproduct;      }  } |

In above example, findProduct will be using “products” cache, while findSoldProduct has overwritten the cache to be used. Additionally, both of them will use the keyGenerator specified on class level.

<http://websystique.com/spring/spring-4-cacheable-cacheput-cacheevict-caching-cacheconfig-enablecaching-tutorial/>

#### **@Profile :**

#### Spring @Profile allow developers to register beans by condition. For example, register beans based on what operating system (Windows, \*nix) your application is running, or load a database properties file based on the application running in development, test, staging or production environment

public static final String ACTIVE\_PROFILES\_PROPERTY\_NAME = "spring.profiles.active";

System.setProperty(AbstractEnvironment.ACTIVE\_PROFILES\_PROPERTY\_NAME, "dev");

@Profile  its is a class level annotation or method level annotation:

@Bean

@Profile("dev")

public CacheManager concurrentMapCacheManager() {

log.debug("Cache manager is concurrentMapCacheManager");

return new ConcurrentMapCacheManager("movieFindCache");

}

@Bean

@Profile("live")

public CacheManager cacheManager() {

log.debug("Cache manager is ehCacheCacheManager");

return new EhCacheCacheManager(ehCacheCacheManager().getObject());

}

@Configuration

@Profile("dev")

public class CacheConfigLive {}

web.xml

<context-param>

<param-name>spring.profiles.active</param-name>

<param-value>live</param-value>

</context-param>

public class MyWebInitializer extends

AbstractAnnotationConfigDispatcherServletInitializer {

//...

@Override

public void onStartup(ServletContext servletContext) throws ServletException {

super.onStartup(servletContext);

servletContext.setInitParameter("spring.profiles.active", "live");

}}

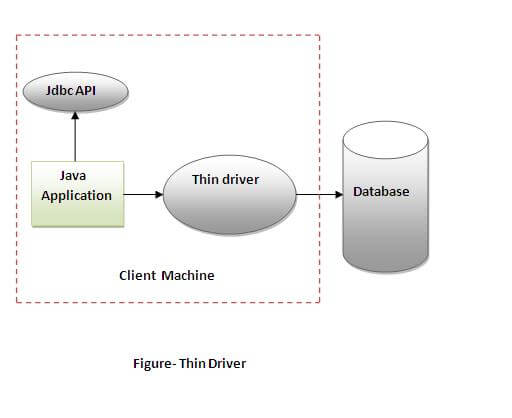
**You can enable multiple profiles.**

@ActiveProfiles({"dev", "mysql","integration"})

**JDBC**

### **Thin driver**

|  |
| --- |
| The thin driver converts JDBC calls directly into the vendor-specific database protocol. That is why it is known as thin driver. It is fully written in Java language. |



### **Advantage:**

* Better performance than all other drivers.
* No software is required at client side or server side.

### **Disadvantage:**

* Drivers depend on the Database.

There are 5 steps to connect any java application with the database using JDBC. These steps are as follows:

* Register the driver class
* Creating connection
* Creating statement
* Executing queries
* Closing connection

The **forName()** method of Class class is used to register the driver class. This method is used to dynamically load the driver class.

**Connection Example:**

|  |
| --- |
| public class DBConnection {  public static Connection getConnection() {  Properties props = new Properties();  FileInputStream fis = null;  Connection con = null;  try {  fis = new FileInputStream("db.properties");  props.load(fis);  // load the Driver Class  Class.forName(props.getProperty("DB\_DRIVER\_CLASS"));  // create the connection now  con = DriverManager.getConnection(props.getProperty("DB\_URL"),  props.getProperty("DB\_USERNAME"),  props.getProperty("DB\_PASSWORD"));  } catch (IOException | ClassNotFoundException | SQLException e) {  // TODO Auto-generated catch block  e.printStackTrace();  }  return con;  }  }  **Excute query:**  try(Connection con = DBConnection.getConnection();  Statement stmt = con.createStatement();  ResultSet rs = stmt.executeQuery(QUERY)) {    while(rs.next()){  int id = rs.getInt("id");  String name = rs.getString("name");  String email = rs.getString("email");  String country = rs.getString("country");  String password = rs.getString("password");  System.out.println(id + "," +name+ "," +email+ "," +country+ "," +password);  }  } catch (SQLException e) {  e.printStackTrace(); }  }  }  **Update Statement:**  PreparedStatement st = connection.prepareStatement(  "Update todo set user=?, desc=?, target\_date=?, is\_done=? where id=?");  st.setString(1, todo.getUser());  st.setString(2, todo.getDesc());  st.setTimestamp(3, new Timestamp(  todo.getTargetDate().getTime()));  st.setBoolean(4, todo.isDone());  st.setInt(5, todo.getId());  st.execute();  st.close();  connection.close();  **Execute query prepare Statement:**  PreparedStatement st = connection.prepareStatement(  "SELECT \* FROM TODO where id=?");  st.setInt(1, id);  ResultSet resultSet = st.executeQuery();  if (resultSet.next()) {  Todo todo = new Todo();  todo.setId(resultSet.getInt("id"));  todo.setUser(resultSet.getString("user"));  todo.setDesc(resultSet.getString("desc"));  todo.setTargetDate(resultSet.getTimestamp("target\_date"));  return todo;  }  st.close();  connection.close(); |

**JDBC API** provides 3 different interfaces to execute the different types of SQL queries. They are,

1) [**Statement**](http://javaconceptoftheday.com/java-jdbc-tutorial-sql-create-insert-select-update-delete-examples/): Used to execute normal SQL queries.

2) **[PreparedStatement](http://javaconceptoftheday.com/preparedstatement-in-java/" \t "_blank)**: Used to execute dynamic or parameterized SQL queries.

PreparedStatement is used to execute dynamic or parameterized SQL queries. PreparedStatement extends Statement interface. You can pass the parameters to SQL query at run time using this interface. It is recommended to use PreparedStatement if you are executing a particular SQL query multiple times. It gives better performance than Statement interface. Because, PreparedStatement are precompiled and the query plan is created only once irrespective of how many times you are executing that query. This will save lots of time

3) **[CallableStatement](http://javaconceptoftheday.com/java-callablestatement-stored-procedures-in-out-parameters-examples/" \t "_blank)*:***  Used to execute the stored procedures.

## Statement Vs PreparedStatement Vs CallableStatement

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Statement** | **PreparedStatement** | **CallableStatement** | | It is used to execute normal SQL queries. | It is used to execute parameterized or dynamic SQL queries. | It is used to call the stored procedures. | | It is preferred when a particular SQL query is to be executed only once. | It is preferred when a particular query is to be executed multiple times. | It is preferred when the stored procedures are to be executed. | | You cannot pass the parameters to SQL query using this interface. | You can pass the parameters to SQL query at run time using this interface. | You can pass 3 types of parameters using this interface. They are – IN, OUT and IN OUT. | | This interface is mainly used for DDL statements like CREATE, ALTER, DROP etc. | It is used for any kind of SQL queries which are to be executed multiple times. | It is used to execute stored procedures and functions. | | The performance of this interface is very low. | The performance of this interface is better than the Statement interface (when used for multiple execution of same query). | The performance of this interface is high. | |

***Spring Jdbc:***

Spring **JdbcTemplate** is a powerful mechanism to connect to the database and execute SQL queries. It internally uses JDBC api, but eliminates a lot of problems of JDBC API.

## Problems of JDBC API

The problems of JDBC API are as follows:

* We need to write a lot of code before and after executing the query, such as creating connection, statement, closing resultset, connection etc.
* We need to perform exception handling code on the database logic.
* We need to handle transaction.
* Repetition of all these codes from one to another database logic is a time consuming task.

## Advantage of Spring JdbcTemplate

Spring JdbcTemplate eliminates all the above mentioned problems of JDBC API. It provides you methods to write the queries directly, so it saves a lot of work and time.

It takes care of creation and release of resources such as creating and closing of connection object etc. So it will not lead to any problem if you forget to close the connection.

It handles the exception and provides the informative exception messages by the help of exception classes defined in the **org.springframework.dao** package.

We can perform all the database operations by the help of JdbcTemplate class such as insertion, updation, deletion and retrieval of the data from the database.

Let us see the methods of spring JdbcTemplate class.

|  |  |  |
| --- | --- | --- |
| **No.** | **Method** | **Description** |
| 1) | public int update(String query) | is used to insert, update and delete records. |
| 2) | public int update(String query,Object... args) | is used to insert, update and delete records using PreparedStatement using given arguments. |
| 3) | public void execute(String query) | is used to execute DDL query. |
| 4) | public T execute(String sql, PreparedStatementCallback action) | executes the query by using PreparedStatement callback. |
| 5) | public T query(String sql, ResultSetExtractor rse) | is used to fetch records using ResultSetExtractor. |
| 6) | public List query(String sql, RowMapper rse) | is used to fetch records using RowMapper. |

# **PreparedStatement:**

We can execute parameterized query using Spring JdbcTemplate by the help of execute() method of JdbcTemplate class. To use parameterized query, we pass the instance of PreparedStatementCallback in the execute method.

|  |
| --- |
| public Boolean saveEmployeeByPreparedStatement(final Employee e){  String query="insert into employee values(?,?,?)";  return jdbcTemplate.execute(query,new PreparedStatementCallback<Boolean>(){  @Override  public Boolean doInPreparedStatement(PreparedStatement ps)  throws SQLException, DataAccessException {    ps.setInt(1,e.getId());  ps.setString(2,e.getName());  ps.setFloat(3,e.getSalary());  return ps.execute();    }  });  } |

# **NamedParameterJdbcTemplate:**

Spring provides another way to insert data by named parameter. In such way, we use names instead of ?(question mark). So it is better to remember the data for the column.

**insert into employee values (:id,:name,:salary)**

**Sping JdbcTemplate Configuration:**

|  |
| --- |
| *@Configuration*  *@EnableTransactionManagement*  *@PropertySource("classpath:database.properties")*  *public class DBConfig {*  *@Autowired*  *private Environment env;*  *@Bean*  *public DataSource getDataSource() {*  *BasicDataSource dataSource = new BasicDataSource();*  *dataSource.setDriverClassName(env.getProperty("database.driverClassName"));*  *dataSource.setUrl(env.getProperty("database.url"));*  *dataSource.setUsername(env.getProperty("database.username"));*  *dataSource.setPassword(env.getProperty("database.password"));*  *return dataSource;*  *}*  *@Bean*  *public JdbcTemplate jdbcTemplate() {*  *return new JdbcTemplate(getDataSource());*  *}*  *@Bean(name="transactionManager")*  *public PlatformTransactionManager txManager() {*  *DataSourceTransactionManager transactionManager = new DataSourceTransactionManager(getDataSource());*  *return transactionManager;*  *}*  *}*  ***Xml Configuration:***  *<context:property-placeholder location="classpath:database.properties"/>*  *<bean id="basicDataSource" class="org.apache.commons.dbcp2.BasicDataSource" destroy-method="close">*  *<property name="driverClassName" value="${database.driverClassName}" />*  *<property name="url" value="${database.url}" />*  *<property name="username" value="${database.username}" />*  *<property name="password" value="${database.password}" />*  *</bean>*  *<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">*  *<property name="dataSource" ref="basicDataSource" />*  *</bean>*  *<bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">*  *<property name="dataSource" ref="basicDataSource" />*  *</bean>*  *<tx:annotation-driven transaction-manager="transactionManager" />* |

**RowMapper:**

Spring JDBC provides RowMapper interface that is used to map row with a Java object. We need to create our own class implementing RowMapper interface to map row with Java object. Find the sample code to implement RowMapperinterface.

### **RowMapper Interface:**

RowMapper interface allows to map a row of the relations with the instance of user-defined class. It iterates the ResultSet internally and adds it into the collection. So we don't need to write a lot of code to fetch the records as ResultSetExtractor.

#### **Advantage of RowMapper over ResultSetExtractor**

RowMapper saves a lot of code becuase it internally adds the data of ResultSet into the collection.

import java.sql.ResultSet;

import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class ArticleRowMapper implements RowMapper<Article> {

@Override

public Article mapRow(ResultSet row, int rowNum) throws SQLException {

Article article = new Article();

article.setArticleId(row.getInt("articleId"));

article.setTitle(row.getString("title"));

article.setCategory(row.getString("category"));

return article;

}

}

We can use our ArticleRowMapper with JdbcTemplate as given below.

public List<Article> getAllArticles() {

String sql = "SELECT articleId, title, category FROM articles";

RowMapper<Article> rowMapper = new ArticleRowMapper();

return this.jdbcTemplate.query(sql, rowMapper);

}

Spring JDBC provides BeanPropertyRowMapper that implements RowMapper. We can directly use it in place of custom RowMapper. We use BeanPropertyRowMapper in the scenario when database table column names and our class fields name are of same. Then we can change above code as following.

public List<Article> getAllArticles() {

String sql = "SELECT articleId, title, category FROM articles";

RowMapper<Article> rowMapper = new BeanPropertyRowMapper<Article>(Article.class);

return this.jdbcTemplate.query(sql, rowMapper);

}

### **JdbcTemplate : Run SQL Queries:**

JdbcTemplate provides methods to run DML and DDL SQL queries. Find the example of some of them.

**a. JdbcTemplate.queryForObject**:

<T> T queryForObject(String sql, RowMapper<T> rowMapper, Object... args)

This method fetches data for a given SQL query as an object using RowMapper. SQL query can have bind parameters. Find the description of parameters.   
**sql**: SQL containing bind parameter.   
**rowMapper**: Object of RowMapper implemented class. RowMapper will map one object per row.   
**args**: Arguments that bind to the query.   
  
**b. JdbcTemplate.query**:

<T> List<T> query(String sql,RowMapper<T> rowMapper)

This method executes static query and maps rows to Java objects using RowMapper. Find the description of parameters.   
**sql**: SQL query to execute.   
**rowMapper**: Object of RowMapper implemented class. RowMapper will map one object per row.   
  
**c. JdbcTemplate.update**:

int update(String sql, Object... args)

This method executes insert, update and delete statements. Find the description of parameters.   
**sql**: SQL containing bind parameter.   
**args**: Arguments that bind to the query.   
  
Now we will perform CREATE, READ, UPDATE and DELETE (CRUD) operation.   
**1. CREATE**: Find sample code snippet for CREATE operation.

public void addArticle(Article article) {

String sql = "INSERT INTO articles (articleId, title, category) values (?, ?, ?)";

jdbcTemplate.update(sql, article.getArticleId(), article.getTitle(), article.getCategory());

}

**2. READ**: Find sample code snippet for READ operation.

public List<Article> getAllArticles() {

String sql = "SELECT articleId, title, category FROM articles";

RowMapper<Article> rowMapper = new BeanPropertyRowMapper<Article>(Article.class);

return this.jdbcTemplate.query(sql, rowMapper);

}

The above method will return a list of objects. If we want to fetch a single object, we can write READ operation code as follows.

public Article getArticleById(int articleId) {

String sql = "SELECT articleId, title, category FROM articles WHERE articleId = ?";

RowMapper<Article> rowMapper = new BeanPropertyRowMapper<Article>(Article.class);

Article article = jdbcTemplate.queryForObject(sql, rowMapper, articleId);

return article;

}

**3. UPDATE**: Find sample code snippet for UPDATE operation.

public void updateArticle(Article article) {

String sql = "UPDATE articles SET title=?, category=? WHERE articleId=?";

jdbcTemplate.update(sql, article.getTitle(), article.getCategory(), article.getArticleId());

}

**4. DELETE**: Find sample code snippet for DELETE operation.

public void deleteArticle(int articleId) {

String sql = "DELETE FROM articles WHERE articleId=?";

jdbcTemplate.update(sql, articleId);

}

**Transaction Management**

Transaction is a Group of related operations as single unit, which should be executed all or none (do not execute).

These actions should either complete entirely or take no effect at all

**Characteristics of Transaction(ACID):**

ACID stands for Atomic, Consistent, isolated and Durable.

1. **AutoMic:**

A transaction should be treated as a single unit of operation, which means the entire sequence of operations is either successful or unsuccessful. i.e. If one of the operation is fails that the others operation are canceled rolled back.

1. **Consistency:**

End of the transaction the system should be in a be a valid state.

1. **Isolated:**

There may be many transaction processing with the same data set at the same time. Each transaction should be isolated from others to prevent data corruption.

The transaction allows multiple people to work on same data in way that one transaction does not effect on the data of rest of the system. Therefore two transaction are can occure simultaneously without dirty reads.

1. **Durable:** the changes of the transactions are persisted to a permeant storage

**Problems:**

* **Dirty Read:**

*Dirty read occurs when one transaction is changing the record, and the other transaction can read this record before the first transaction has been committed or rolled back. This is known as a dirty read scenario because there is always the possibility that the first transaction may rollback the change, resulting in the second transaction having read an invalid data.*

***Dirty Read Example:-***

*Transaction A begins. UPDATE EMPLOYEE SET SALARY = 10000 WHERE EMP\_ID= ‘123’;*

*Transaction B begins.  
SELECT \* FROM EMPLOYEE;  
(Transaction B sees data which is updated by transaction A. But, those updates have not yet been committed.)*

* **Non-Repeatable Read:-**

*Non Repeatable Reads happen when in a same transaction same query yields to a different result. This occurs when one transaction repeatedly retrieves the data, while a difference transactions alters the underlying data. This causes the different or non-repeatable results to be read by the first transaction.*

***Non-Repeatable Example:***

*Transaction A begins.  
SELECT \* FROM EMPLOYEE WHERE EMP\_ID= ‘123’;*

*Transaction B begins.  
UPDATE EMPLOYEE SET SALARY = 20000 WHERE EMP\_ID= ‘123’;  
(Transaction B updates rows viewed by the transaction A before transaction A commits.) If Transaction A issues the same SELECT statement, the results will be different.*

* **Phantom Read:**

*Phantom read occurs where in a transaction execute same query more than once, and the second transaction result set includes rows that were not visible in the first result set. This is caused by another transaction inserting new rows between the execution of the two queries. This is similar to a non-repeatable read, except that the number of rows is changed either by insertion or by deletion.*

***Phantom Read Example:-***

*Transaction A begins.  
SELECT \* FROM EMPLOYEE WHERE SALARY > 10000 ;*

*Transaction B begins.  
INSERT INTO EMPLOYEE (EMP\_ID, FIRST\_NAME, DEPT\_ID, SALARY) VALUES (‘111’, ‘Jamie’, 10, 35000);  
Transaction B inserts a row that would satisfy the query in Transaction A if it were issued again.*

**Isolation Levels:**

* **TRANSACTION\_READ\_UNCOMMITTED -1**

*Before committing the transaction its intermediate results can be used by other transactions*

*Internally will not use any locks.*

*It does not prevent (check/dirty) dirty read problem, Non-Repeatable Read Problem and Phantom Read problem.*

*We can use this isolation level just to indicate database supports transactions.*

*This. Isolation level is n’t recommended to use.*

* **TRANSACTION\_READ\_COMMITTED - 2:**

*Read the transactions only committed data, un-committed data not allowed to read other transaction.*

*Before committing the data other transactions are not allowed to read/update*

*Only committed data transaction can read.*

*In this case dirty read problem prevent.*

*Allowed non repeatable read problem*

*Phantom read problem allowed*

* **TRANSACTION\_REPEATABLE\_READ – 4**

*This is Default isolation level most of the databases, internally the results of SQL query will be locked for only one transaction, if we perform multiple read operation, then there is guarantee that for getting same result.*

*Transaction can perform multiple time we can get results same every time*

*Select \* form emp where sal > 1000*

*Result of sql query will be locked for only transaction at time, so any other transaction not allowed (access/updated) perform that data, biggest advantage is other transactions cannot modified, because of that transaction get same results every time.*

*In this case dirty read problem not allowed.*

*repeatable read problem not Allowed*

*Phantom read problem is allowed (means insertion is possible)*

* **TRANSACTION\_SERIALIZABLE-8**

*It is the highest level of isolation.*

*The total table will be locked for one transaction at time.*

*It prevent Dirty Read, Non-Repeatable read and phantom read problem.*

*Not recommended to use because it may creates performance problems.*

***Note.***

*Con.getTransactionIsolationLevel() //result 2*

*Con.setTransactionIsolutionLevel(2)*

*Default in databases:*

*Oracle TRANSACTION\_READ\_COMMITTED-2 // supported 2 and 8*

*Mysql TRANSACTION\_REPEATABLE\_READ-4 // supported all isolation level 1,2,4,8*

***High-level details:***

|  |  |  |  |
| --- | --- | --- | --- |
|  | *dirty reads* | *non-repeatable reads* | *phantom reads* |
| *READ\_UNCOMMITTED* | *yes* | *yes* | *yes* |
| *READ\_COMMITTED* | *no* | *yes* | *yes* |
| *REPEATABLE\_READ* | *no* | *no* | *yes* |
| *SERIALIZABLE* | *no* | *no* | *no* |

**Logging and Caching mechanism:**

**Logging mechanism in spring boot:**

We implemented SLF4J in GPR3 application as it acts as a façade for different logging frameworks (e.g. java.util.logging, logback, Log4j). It offers a generic API making the logging independent of the actual implementation. This allows for different logging frameworks to coexist. It also helps migrate from one framework to another. It is simple, yet flexible, and allows for readability and performance improvements.

**Management and Monitoring using JMX:**

We created LoggerHandler java class to handle log levels through jconsole at run time so that we do not have to re-run the application to switch between the log levels.

**Cache Implementation (Ehcache) in spring boot:**

We implemented Ehcache for caching in GPR3 application. Ehcache is Fast & LightWeight, provides scalability into terabytes, extensible, flexible and also Ehcache has full support for high performance distributed caching.

**Management and Monitoring using JMX:**

We created EhcacheHandler java class to handle caching operations (enableCache, disableCache, clearCache, clearAllCachem deleteCache) at run time via jconsole without restarting the application.

https://www.concretepage.com/spring-5/spring-jdbc-example

*Cloud*

*https://o7planning.org/en/11723/understanding-spring-cloud-config-server-with-example*

[*https://www.dineshonjava.com/spring-mvc-internalresourceviewresolver-configuration-example/*](https://www.dineshonjava.com/spring-mvc-internalresourceviewresolver-configuration-example/)

*https://www.24tutorials.com/spark/caching-persistence-apache-spark/*

*http://timepasstechies.com/spark-rdd-api-transformations-actions-tutorial-examples-part-1/*

*https://saurzcode.in/2015/10/what-is-rdd-in-spark-and-why-do-we-need-it/*

*http://www.agildata.com/apache-spark-rdd-vs-dataframe-vs-dataset/*

*https://acadgild.com/blog/spark-sql-for-e-commerce-data-analysis/*

*spring*

*======*

*http://websystique.com/spring/spring-dependency-injection-annotation-beans-auto-wiring-using-autowired-qualifier-resource-annotations-configuration/*

*https://springframework.guru/spring-framework-annotations/*

*http://javabeginnerstutorial.com/spring-framework-tutorial/spring-annotation-example/*

[*https://howtodoinjava.com/category/spring5/*](https://howtodoinjava.com/category/spring5/)

*logging*

*==*

*Wrote test cases using* spring test and DBUnit framework.

[*https://www.dev2qa.com/spring-boot-integrates-slf4j-and-logback/*](https://www.dev2qa.com/spring-boot-integrates-slf4j-and-logback/)