Spring Framework

By

Praveen Naga

Blog: https://praveennagatech.blogspot.com

Facebook Group: https://www.facebook.com/groups/268426377837151

Github repo: https://github.com/praveennaga

Email: opraveennaga@gmail.com

Buy me a Cup of Coffee:
https://docs.google.com/forms/d/e/1FAIpQLSfZxtgAxzMv83uwdEXezVFfL
NSKlQgu7eDcMOeGtS2cRnOBrA/viewform?vc=0&c=0&w=1

Spring Introduction

History of Spring

1995 — Java Applets — Gaming
1996 — Java Beans — Business Logic
1998 — EJB — Business Logic + Services
2003 — Spring — Business Logic + Services(Extra Advantages)

Previous name of Spring is Interface 21 and this was introduced by Rod Johnson.

Previously maintained by spring org but now it is Pivotal.

Spring is an alternative for EJB as EJB depends on application servers and it is tightly coupled.

EJB promised scalability,security and high availability.

EJB on paper looks good but

1. Its takes time to configure
2. Error prone
3. Tightly coupled to the application server, making it difficult to test

- ✓ Spring is a loosely coupled and it's lightweight as it doesn't depend on OS where as it just uses JDK.
- ✓ Spring is a noninvasive, opensource and lightweight framework.
- ✓ Spring is implemented based on runtime polymorphism and Has-A relationship.

Invasive framework will force a developer to extend a framework or implement a framework interface while creating projects.
For example, Struts framework is invasive.

Non-invasive framework does not force developers to extend a framework class or implement a framework interface.
For example, Hibernate and spring are non-invasive framework.

Advantages of Spring
1. Lightweight – We can pick and use the necessary modules required for our project.
2. Flexible – Use DI and configured by XML or Annotation based style.
3. Loosely coupled – Use DI
4. Easy to test – provides testing modules that makes testing easier.

What is Spring IOC Container?

Spring IOC container is used for creating and injecting dependencies automatically.

Spring IOC

1.creates the objects

2.wire them together

3.configure them

4.Manage their complete lifecycle from creation till destruction

5.Uses dependency injection to manage the components that make up an application

In Spring we have the below containers

a) IOC- BeanFactory and ApplicationContext.

b) MVC Web container developed on top of IOC – WebApplicationContext.

1. **BeanFactory** (I) – XMLBeanFactory(C)

2. **ApplicationContext**(I) -ConfigurableApplicationContext(I)

a) ClasspathXMLApplicationContext(C)

b) FileSystemXMLApplicationContext(C)

c) AnnotationConfigApplicationContext(C)

WebApplicationContext (I)

a) WebApplicationContextUtilFactory(C)

b) AnnotationConfigWebApplicationContext(C)

Points to remember

- ✓ By default bean is singleton i.e.. Single object is created even though we call getBean() multiple times.

- ✓ If we declare scope as prototype for every request we will get new object.

- ✓ Spring can access private constructor also

IOC vs DI

IOC(Inversion of Control) -> Giving control to the container to get an instance of the object means instead you creating an object using new operator, let the controller do it for you.

DI(Dependency Injection)-> It is a form of IOC where implementations are passed into an object through constructors/setters which the object will depend on in order to behave correctly.

Spring Bean Scopes
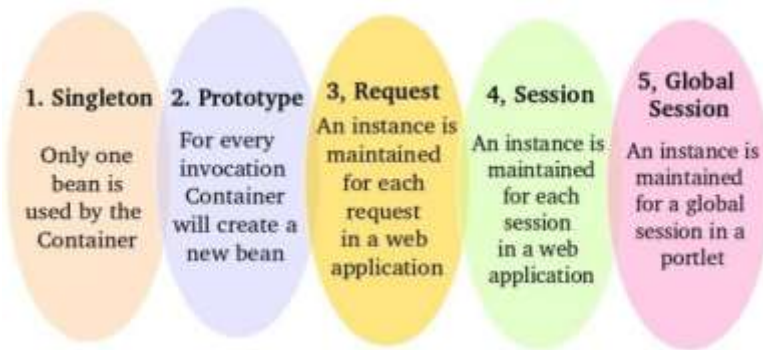
1. Singleton

- ✓ It is the default scope of a bean in spring container.
- ✓ In this scope, spring creates only one instance of the bean and it is served for every request for that bean from cache with in the container.
- ✓ Only one instance of bean per bean id.
- ✓ This differs from the java singleton design pattern which is based on per classloader where as in springs, singleton is per container and per bean.
- ✓ Singleton scope id used for stateless beans.

2. Prototype

- ✓ Spring container creates new instance of bean for each and every request for that bean.
- ✓ Prototype scope is useful for stateful bean.
- ✓ After instantiating bean ad submitting it to client, spring container does not maintain the record.so to release the resources held by prototype scope bean, client should implement custom bean post processor.

Let's see an example related to Singleton and Prototype bean scopes

```
1  package com.praveen.spring.core.beanscope;
2
3  public class TokenMachine {
4
5  }
```

```
1  package com.praveen.spring.core.beanscope;
2
3  public class Token {
4
5  }
```

```java
 1  package com.praveen.spring.core.beanscope;
 2
 3  import org.springframework.context.annotation.Bean;
 4  import org.springframework.context.annotation.Configuration;
 5  import org.springframework.context.annotation.Scope;
 6
 7  @Configuration
 8  public class AppConfig {
 9
10      @Bean(name = "token")
11      @Scope(scopeName = "prototype")
12      public Token getToken() {
13          return new Token();
14      }
15
16      @Bean(name = "tokenMachine")
17      @Scope(scopeName = "singleton")
18      public TokenMachine getTokenMachine() {
19          return new TokenMachine();
20      }
21
22  }
```

```java
 1  package com.praveen.spring.core.beanscope;
 2
 3  import org.springframework.context.annotation.AnnotationConfigApplicationContext;
 4  import org.springframework.context.support.AbstractApplicationContext;
 5
 6  public class AnnotatedBeanScopeTest {
 7      public static void main(String[] args) {
 8          AbstractApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
 9          TokenMachine tokenMachine1 = (TokenMachine) context.getBean("tokenMachine");
10          TokenMachine tokenMachine2 = (TokenMachine) context.getBean("tokenMachine");
11          boolean areTokenMachineInstancesSame = tokenMachine1.equals(tokenMachine2);
12          System.out.println(areTokenMachineInstancesSame); // print true
13          Token token1 = (Token) context.getBean("token");
14          Token token2 = (Token) context.getBean("token");
15          boolean areTokenInstancesSame = token1.equals(token2);
16          System.out.println(areTokenInstancesSame); // prints false
17          context.close();
18      }
19  }
```

Output

```
Oct 08, 2019 3:19:44 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb0d902: startup date [Tue Oct 08 15:19:44 IST 2019]; root of context hierarch
true
false
Oct 08, 2019 3:19:44 PM org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb0d902: startup date [Tue Oct 08 15:19:44 IST 2019]; root of context hierarchy
```

Spring Bean scope features

Use the prototype bean when the bean carries a state i.e. its stateful. Use singleton bean for stateless beans.

For a prototype bean, the container does not manage the complete lifecyle of the bean. It initiates, configures and assembles a prototype bean and then hands it over to the client. The destruction lifecycle methods are not called.

The dependencies of a bean are resolved at instantiation time, therefore if a singleton bean needs a prototype bean, the container creates one during instantiation of the singleton bean. The singleton bean would therefore always have the same instance of prototype bean.

3.Request

request scope bean is used in HTTP request life cycle.

For each and every request, new instance of bean is created and is alive for complete HTTP request life cycle.

@Bean

@Scope(value = WebApplicationContext.SCOPE_REQUEST, proxyMode = ScopedProxyMode.TARGET_CLASS)

public HelloMessageGenerator requestMessage() {

return new HelloMessageGenerator();

}

The attribute proxyMode is needed as during the instantiation of web application context, there is no active request. The Spring Framework will create the proxy for the injection as a dependency.

Next, you should define the controller that has a reference to the requestMessage bean. For that, you need to access the same request more than once to test the web-specific scopes.

@Controller

public class ScopesController {

@Resource(name = "requestMessage")

HelloMessageGenerator requestMessage;

@RequestMapping("/scopes")

public String getScopes(Model model) {

requestMessage.setMessage("Good morning!");

model.addAttribute("requestMessage", requestMessage.getMessage());

return "scopesExample";

}

}

4.Session

it is also related to web applications. On every session, creates new object in container.

@Bean

@Scope(value = WebApplicationContext.SCOPE_SESSION, proxyMode = ScopedProxyMode.TARGET_CLASS)

public HelloMessageGenerator sessionMessage() {

return new HelloMessageGenerator();

}

@Controller

public class ScopesController {

@Resource(name = "sessionMessage")

HelloMessageGenerator sessionMessage;

@RequestMapping("/scopes")

public String getScopes(Model model) {

sessionMessage.setMessage("Hello there!");

model.addAttribute("sessionMessage", sessionMessage.getMessage());

return "scopesExample";

}

}

5.**globalsession**

This scopes the bean definition to global HTTP session and is only valid for web-aware Application context of Spring. This type is used in portlet container applications where each portlet having their session.

@Bean

@Scope(value = WebApplicationContext.SCOPE_GLOBAL_SESSION, proxyMode = ScopedProxyMode.TARGET_CLASS)

public HelloMessageGenerator globalSessionMessage() {

return new HelloMessageGenerator();

}

Spring Bean Lifecycle

Life of traditional java objects starts on calling new operator which instantiates the object and finalize() method is getting called when the object is eligible for garbage collection. Life cycle of Spring beans are different as compared to traditional java objects.

Spring framework provides the following ways which can be used to control the lifecycle of bean:
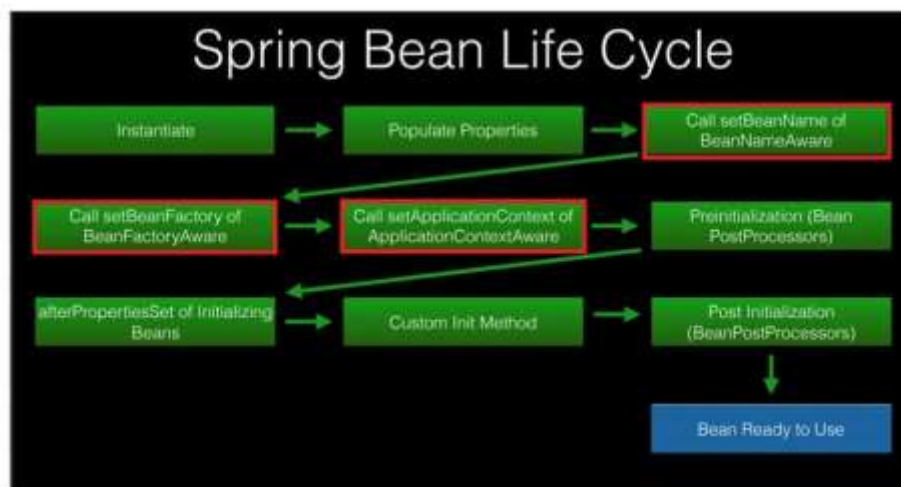
InitializingBean and DisposableBean callback interfaces
Bean Name, bean factory and Application Context Aware interfaces for specific behavior
custom init() and destroy() methods in bean configuration file

For annotation based configurations
@PostConstruct and @PreDestroy annotations

Following is sequence of a bean lifecycle in Spring:

Following diagram shows the method calling at the time of destruction.

- ✓ **Instantiate**: First the spring container finds the bean's definition from the XML file and instantiates the bean.
- ✓ **Populate properties**: Using the dependency injection, spring populates all of the properties as specified in the bean definition.
- ✓ **Set Bean Name**: If the bean implements BeanNameAware interface, spring passes the bean's id to setBeanName() method.
- ✓ **Set Bean factory**: If Bean implements BeanFactoryAware interface, spring passes the beanfactory to setBeanFactory() method.
- ✓ **Pre Initialization**: Also called post process of bean. If there are any bean BeanPostProcessors associated with the bean, Spring calls postProcesserBeforeInitialization() method.

- ✓ **Initialize beans**: If the bean implements IntializingBean,its afterPropertySet() method is called. If the bean has init method declaration, the specified initialization method is called.
- ✓ **Post Initialization**: – If there are any BeanPostProcessors associated with the bean, their postProcessAfterInitialization() methods will be called.
- ✓ **Ready to use**: Now the bean is ready to use by the application
- ✓ **Destroy**: If the bean implements DisposableBean , it will call the destroy() method
- ✓ **ApplicationContextAware**: ApplicationContextAware can be used as an alternative for lookup method injection when a prototype bean is injected to a singleton bean

Let's consider an example

```java
package com.praveen.spring.core.beanlifecycle.beanscope;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("prototype")
public class RequestHandler {

    RequestHandler() {
        System.out.println("In Request Handler Constructor");
    }

    public void handleRequest() {
        System.out.println("Handling request");
    }
}
```

```java
package com.praveen.spring.core.beanlifecycle.beanscope;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("singleton")
public class RequestManager implements ApplicationContextAware{
    private RequestHandler requestHandler;
    private ApplicationContext applicationContext;
    public void handleRequest(){
      requestHandler = getRequestHandler();
      requestHandler.handleRequest();
    }
    // method to return new instance
    public RequestHandler getRequestHandler() {
      return applicationContext.getBean("requesthandler", RequestHandler.class);
    }

    @Override
    public void setApplicationContext(ApplicationContext applicationContext)
      throws BeansException {
      this.applicationContext = applicationContext;
    }
}
```

```java
package com.praveen.spring.core.beanlifecycle.beanscope;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class RequestConfig {

    @Bean(name="requestmanager")
     public RequestManager requestManager(){
       return new RequestManager();
    }

    @Bean(name="requesthandler")
     public RequestHandler requestHandler(){
       return new RequestHandler();
    }

}
```

```java
package com.praveen.spring.core.beanlifecycle.beanscope;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class BeanScopeTest {
    public static void main( String[] args ){
            AnnotationConfigApplicationContext context= new AnnotationConfigApplicationContext(RequestConfig.class);
            RequestManager bean = (RequestManager) context.getBean("requestmanager");
            bean.handleRequest();
            bean.handleRequest();
            bean.handleRequest();
            context.close();

        }

}
```

Output:

```
In Request Handler Constructor
Handling request
Handling request
Handling request
```

Let's see an example on other bean life cycle methods

```java
package com.praveen.spring.core.beanlifecycle;

import javax.annotation.PostConstruct;

@Component
public class Employee implements InitializingBean, DisposableBean, BeanNameAware, BeanPostProcessor {

    private String empName;
    private int empId;
    private double empSal;

    Employee() {
        System.out.println("Employee Constructor invoked");
    }

    public String getEmpName() {
        return empName;
    }

    public void setEmpName(String empName) {
        this.empName = empName;
    }

    public int getEmpId() {
        return empId;
    }

    public void setEmpId(int empId) {
        this.empId = empId;
    }

    public double getEmpSal() {
        return empSal;
    }
```

```java
    @Override
    public void setBeanName(String name) {
        System.out.println(name + " bean has been initialized.");

    }

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("Inside post process before initialization: " + beanName);
        return bean;
    }

    @PostConstruct
    public void init() throws Exception {
        System.out.println("Post Construct method invoked");
    }

    @Override
    public void afterPropertiesSet() throws Exception {
        System.out.println("After Properties Set method invoked");

    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("Inside post process after initialization: " + beanName);
        return bean;
    }

    @PreDestroy
    public void customDestroy() throws Exception {
        System.out.println("Custom Destroy method invoked");
    }
```

```java
    @Override
    public void destroy() throws Exception {
        System.out.println("Destroy method  invoked");

    }

    @Override
    public String toString() {
        return "Employee [empName=" + empName + ", empId=" + empId + ", empSal=" + empSal + "]";
    }

}
```

```java
package com.praveen.spring.core.beanlifecycle;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class EmployeeTest {

    public static void main(String args[]) {
        AnnotationConfigApplicationContext context= new AnnotationConfigApplicationContext(EmployeeConfig.class);
        Employee emp= (Employee)context.getBean("employee");
        System.out.println(emp);
        //Shut down event is fired when context.close() or context.registerShutdownHook() is invoked.
        context.close();
    }
}
```

```java
package com.praveen.spring.core.beanlifecycle;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class EmployeeConfig {

    @Bean(name="employee")
    public Employee newEmployee() {
        Employee emp= new Employee();
        emp.setEmpId(149903);
        emp.setEmpName("Praveen");
        emp.setEmpSal(100000000d);

        return emp;

    }

}
```

Output

```
Oct 08, 2019 3:37:38 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb0d902: startup date [Tue Oct 08 15:37:38 IST 2019]; root of context hierarch
Oct 08, 2019 3:37:39 PM org.springframework.context.support.PostProcessorRegistrationDelegate$BeanPostProcessorChecker postProcessAfterInitialization
INFO: Bean 'employeeConfig' of type [com.praveen.spring.core.beanlifecycle.EmployeeConfig$$EnhancerBySpringCGLIB$$86d3a5] is not eligible for getting processed by all Bea
Employee Constructor invoked
employee bean has been initialized.
Post Construct method invoked
After Properties Set method invoked
Inside post process before initialization: org.springframework.context.event.internalEventListenerProcessor
Inside post process after initialization: org.springframework.context.event.internalEventListenerProcessor
Inside post process before initialization: org.springframework.context.event.internalEventListenerFactory
Inside post process after initialization: org.springframework.context.event.internalEventListenerFactory
Employee [empName=Praveen, empId=149903, empSal=1.0E8]
Oct 08, 2019 3:37:39 PM org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb0d902: startup date [Tue Oct 08 15:37:38 IST 2019]; root of context hierarchy
Custom Destroy method invoked
Destroy method  invoked
```

Spring Dependency Injection

DI(Dependency Injection):

It is a form of IOC where implementations are passed into an object through constructors/setters which the object will depend on in order to behave correctly.

1. Setter Injection

Spring Setter Injection is nothing but injecting the Bean Dependencies using the Setter methods on an Object.

Unlike Spring Constructor Injection, in Setter Injection, the object is created first and then the dependency is injected.

For example,

XML based injection

```xml
beanssetter.xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
3  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
4  <beans>
5      <bean id="department"
6          class="com.praveen.spring.core.di.setter.xml.SIDepartment">
7          <property name="deptName" value="Tech" />
8          <property name="deptId" value="1000" />
9      </bean>
10     <bean id="employee"
11         class="com.praveen.spring.core.di.setter.xml.SIEmployee">
12         <property name="empName" value="Praveen" />
13         <property name="empId" value="149903" />
14         <property name="empSal" value="10000000" />
15         <property name="dept" ref="department" />
16     </bean>
17 </beans>
```

```java
package com.praveen.spring.core.di.setter.xml;

public class SIDepartment {

    private String deptName;
    private int deptId;

    public SIDepartment() {

    }

    public String getDeptName() {
        return deptName;
    }

    public void setDeptName(String deptName) {
        this.deptName = deptName;
    }

    public int getDeptId() {
        return deptId;
    }

    public void setDeptId(int deptId) {
        this.deptId = deptId;
    }

}
```

```java
package com.praveen.spring.core.di.setter.xml;

public class SIEmployee {

    private String empName;
    private int empId;
    private double empSal;
    private SIDepartment dept;

    public SIEmployee() {

    }

    public String getEmpName() {
        return empName;
    }

    public void setEmpName(String empName) {
        this.empName = empName;
    }

    public int getEmpId() {
        return empId;
    }

    public void setEmpId(int empId) {
        this.empId = empId;
    }

    public double getEmpSal() {
        return empSal;
    }

    public void setEmpSal(double empSal) {
        this.empSal = empSal;
    }

    public SIDepartment getDept() {
        return dept;
    }

    public void setDept(SIDepartment dept) {
        this.dept = dept;
    }

}
```

```java
package com.praveen.spring.core.di.setter.xml.test;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;

import com.praveen.spring.core.di.setter.xml.SIEmployee;

public class SetterInjectionTest {

    public static void main(String[] args) {

        // Usage of BeanFactory

        @SuppressWarnings("deprecation")
        BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beanssetter.xml"));

        SIEmployee emp= (SIEmployee)factory.getBean("employee");

        System.out.println("Employee Id :"+emp.getEmpId());
        System.out.println("Employee Name :"+emp.getEmpName());
        System.out.println("Employee Salary :"+emp.getEmpSal());
        System.out.println("Employee Department ID :" + emp.getDept().getDeptId());
        System.out.println("Employee Department Name :" + emp.getDept().getDeptName());

        // Usage of ApplicationContext

        AbstractApplicationContext context= new ClassPathXmlApplicationContext("beanssetter.xml");
        SIEmployee emp1= (SIEmployee)context.getBean("employee");

        System.out.println("Employee1 Id :"+emp1.getEmpId());
        System.out.println("Employee1 Name :"+emp1.getEmpName());
        System.out.println("Employee1 Salary :"+emp1.getEmpSal());
```

```
        System.out.println("Employee1 Salary :"+emp1.getEmpSal());
        System.out.println("Employee1 Department ID :" + emp1.getDept().getDeptId());
        System.out.println("Employee1 Department Name :" + emp1.getDept().getDeptName());

        context.close();

    }

}
```

Output:

```
Oct 08, 2019 4:03:28 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [beanssetter.xml]
Employee Id :149903
Employee Name :Praveen
Employee Salary :1.0E7
Employee Department ID :1000
Employee Department Name :Tech
Oct 08, 2019 4:03:28 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@97e1986: startup date [Tue Oct 08 16:03:28 IST 2019]; root of context hierarchy
Oct 08, 2019 4:03:28 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [beanssetter.xml]
Employee1 Id :149903
Employee1 Name :Praveen
Employee1 Salary :1.0E7
Employee1 Department ID :1000
Employee1 Department Name :Tech
Oct 08, 2019 4:03:29 PM org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@97e1986: startup date [Tue Oct 08 16:03:28 IST 2019]; root of context hierarchy
```

Annotation based injection

We will make use of above SIDepartment.java and SIEmployee.java and will replace the beans.xml with SIEmployeeConfig.java

```java
package com.praveen.spring.core.di.setter.annotation.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.praveen.spring.core.di.setter.annotation.SIDepartment;
import com.praveen.spring.core.di.setter.annotation.SIEmployee;

@Configuration
@ComponentScan(basePackages = {"com.praveen.spring.core.di.setter.annotation"})
public class SIEmployeeConfig {

    @Bean(name="siemployee")
    public SIEmployee siemployee(){
        SIEmployee emp= new SIEmployee();
        SIDepartment dept= new SIDepartment();
        dept.setDeptId(1000);
        dept.setDeptName("Tech");

        emp.setDept(dept);
        emp.setEmpId(149903);
        emp.setEmpName("Praveen");
        emp.setEmpSal(10000000d);
        return emp;
    }
}
```

```
package com.praveen.spring.core.di.setter.annotation.test;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.praveen.spring.core.di.setter.annotation.SIEmployee;
import com.praveen.spring.core.di.setter.annotation.config.SIEmployeeConfig;

public class SetterInjectionTest {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context= new AnnotationConfigApplicationContext(SIEmployeeConfig.class);
        SIEmployee emp1= (SIEmployee)context.getBean("siemployee");
        System.out.println("Employee1 Id :"+emp1.getEmpId());
        System.out.println("Employee1 Name :"+emp1.getEmpName());
        System.out.println("Employee1 Salary :"+emp1.getEmpSal());
        System.out.println("Employee1 Department ID :" + emp1.getDept().getDeptId());
        System.out.println("Employee1 Department Name :" + emp1.getDept().getDeptName());

        context.close();

    }

}
```

Output

```
Oct 08, 2019 4:05:55 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb8d982: startup date [Tue Oct 08 16:05:55 IST 2019]; root of context hierarchy
Employee1 Id :149903
Employee1 Name :Praveen
Employee1 Salary :1.0E7
Employee1 Department ID :0
Employee1 Department Name :null
Oct 08, 2019 4:05:56 PM org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb8d902: startup date [Tue Oct 08 16:05:55 IST 2019]; root of context hierarchy
```

In setter injection, the objects get created first and then the dependency is injected.

In case of Setter Injection, the setter methods are annotated with @Autowired. Spring will first use the no-argument constructor to instantiate the bean and then call setter methods to inject the dependencies.

2. **Constructor based Injection**

It is a type of Spring Dependency Injection, where object's constructor is used to inject dependencies. This type of injection is safer as the objects won't get created if the dependencies aren't available or dependencies cannot be resolved.

For example,

Using XML based configuration

```
X  beansconstructor.xml ✕
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
 3  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
 4  <beans>
 5  <bean id="condepartment"
 6      class="com.praveen.spring.core.di.constructor.xml.CIDepartment">
 7      <constructor-arg value="Tech" />
 8      <constructor-arg value="100" />
 9  </bean>
10  <bean id="conemployee"
11      class="com.praveen.spring.core.di.constructor.xml.CIEmployee">
12      <constructor-arg value="Praveen" />
13      <constructor-arg value="149903" />
14      <constructor-arg value="10000000" />
15      <constructor-arg ref="condepartment" />
16  </bean>
17
18  </beans>
```

```java
package com.praveen.spring.core.di.constructor.xml;

public class CIDepartment {

    private String deptName;
    private int deptId;

    public CIDepartment(String deptName, int deptId) {
        super();
        this.deptName = deptName;
        this.deptId = deptId;
    }
    public String getDeptName() {
        return deptName;
    }

    public int getDeptId() {
        return deptId;
    }


}
```

```java
package com.praveen.spring.core.di.constructor.xml;

public class CIEmployee {

    private String empName;
    private int empId;
    private double empSal;
    private CIDepartment dept;

    public CIEmployee(String empName, int empId, double empSal, CIDepartment dept) {
        super();
        this.empName = empName;
        this.empId = empId;
        this.empSal = empSal;
        this.dept = dept;
    }

    public String getEmpName() {
        return empName;
    }

    public int getEmpId() {
        return empId;
    }

    public double getEmpSal() {
        return empSal;
    }

    public CIDepartment getDept() {
        return dept;
    }

}
```

```java
package com.praveen.spring.core.di.constructor.xml.test;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.xml.XmlBeanFactory;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.core.io.ClassPathResource;

import com.praveen.spring.core.di.constructor.xml.CIEmployee;

public class ConstructionInjectionTest {

public static void main(String[] args) {

        // Usage of BeanFactory

        @SuppressWarnings("deprecation")
        BeanFactory factory = new XmlBeanFactory(new ClassPathResource("beansconstructor.xml"));

        CIEmployee emp= (CIEmployee)factory.getBean("conemployee");

        System.out.println("Employee Id :"+emp.getEmpId());
        System.out.println("Employee Name :"+emp.getEmpName());
        System.out.println("Employee Salary :"+emp.getEmpSal());
        System.out.println("Employee Department ID :" + emp.getDept().getDeptId());
        System.out.println("Employee Department Name :" + emp.getDept().getDeptName());

        // Usage of ApplicationContext

        AbstractApplicationContext context= new ClassPathXmlApplicationContext("beansconstructor.xml");
        CIEmployee emp1= (CIEmployee)context.getBean("conemployee");

        System.out.println("Employee1 Id :"+emp1.getEmpId());
        System.out.println("Employee1 Name :"+emp1.getEmpName());
        System.out.println("Employee1 Salary :"+emp1.getEmpSal());
        System.out.println("Employee1 Department ID :" + emp1.getDept().getDeptId());

        System.out.println("Employee1 Department Name :" + emp1.getDept().getDeptName());

        context.close();

    }

}
```

Output

```
Oct 08, 2019 4:24:33 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [beansconstructor.xml]
Employee Id :149903
Employee Name :Praveen
Employee Salary :1.0E7
Employee Department ID :100
Employee Department Name :Tech
Oct 08, 2019 4:24:34 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@2cb4c3ab: startup date [Tue Oct 08 16:24:33 IST 2019]; root of context hierarchy
Oct 08, 2019 4:24:34 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [beansconstructor.xml]
Employee1 Id :149903
Employee1 Name :Praveen
Employee1 Salary :1.0E7
Employee1 Department ID :100
Employee1 Department Name :Tech
Oct 08, 2019 4:24:34 PM org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@2cb4c3ab: startup date [Tue Oct 08 16:24:33 IST 2019]; root of context hierarchy
```

Using Annotation based configuration

We will make use of above CIDepartment.java and CIEmployee.java and will replace the beans.xml with CIEmployeeConfig.java

```java
package com.praveen.spring.core.di.constructor.annotation.config;

import org.springframework.context.annotation.Bean;

@Configuration
public class CIEmployeeConfig {

    @Bean(name="ciemployee")
    public CIEmployee ciemployee(){
        CIDepartment dept= new CIDepartment("Tech",1000);
        CIEmployee emp= new CIEmployee("Praveen",149903,10000000d,dept);
        return emp;
    }
}
```

```java
package com.praveen.spring.core.di.constructor.annotation.test;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.praveen.spring.core.di.constructor.annotation.CIEmployee;
import com.praveen.spring.core.di.constructor.annotation.config.CIEmployeeConfig;

public class ConstructorInjectionTest {

    public static void main(String[] args) {

        AnnotationConfigApplicationContext context= new AnnotationConfigApplicationContext(CIEmployeeConfig.class);
        CIEmployee emp1= (CIEmployee)context.getBean("ciemployee");
        System.out.println("Employee1 Id :"+emp1.getEmpId());
        System.out.println("Employee1 Name :"+emp1.getEmpName());
        System.out.println("Employee1 Salary :"+emp1.getEmpSal());
        System.out.println("Employee1 Department ID :" + emp1.getDept().getDeptId());
        System.out.println("Employee1 Department Name :" + emp1.getDept().getDeptName());

        context.close();

    }

}
```

Output:

```
Oct 08, 2019 4:26:25 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb0d902: startup date [Tue Oct 08 16:26:25 IST 2019]; root of context hierarc
Employee1 Id :149903
Employee1 Name :Praveen
Employee1 Salary :1.0E7
Employee1 Department ID :1000
Employee1 Department Name :Tech
Oct 08, 2019 4:26:25 PM org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb0d902: startup date [Tue Oct 08 16:26:25 IST 2019]; root of context hierarchy
```

   B l o g :   https://praveennagatech.blogspot.com
   F a c e b o o k   G r o u p :   https://www.facebook.com/groups/268426377837151
   G i t h u b   r e p o :   https://github.com/praveennaga
   E m a i l :   opraveennaga@gmail.com

Notice here that neither the setters nor the no-argument constructors were invoked by Spring.

The dependencies were injected purely by means of Constructors. This approach is preferred over Spring Setter Injection

When constructor is used to set instance variables on an objects, it is called as Constructor Injection

When we go for Setter Injection or when we go for Constructor Injection?

✓ At the time of creating our target class object, the dependent objects are injected or instantiated through Constructor Injection.
Whereas in Setter Injection dependent objects are not injected or instantiated while creating the target class object; those will be injected after the target class has been instantiated by calling the setter on the target object.

✓ If bean class contains only one property or if all the property of bean class should participate in dependency injection then we should go for constructor injection because constructor injection is very faster than setter injection.

✓ If bean class contains more than one property and there is no need of making all properties (means optional to inject) are participating in dependency injection, then we should go for setter injection. So it is bit delayed injection.

✓ Constructor Injection doesn't support cyclic dependency.
Whereas Setter injection supports cyclic or circular dependency injection.

✓ To perform constructor injection on n properties in all angles, n! Constructors are required. we will use <constructor-arg>.
Whereas to perform setter injection on n properties in all angles "n-setter" methods are required. we will use <property> tag.

Spring supports dependency injection on the following properties:
a. Simple properties (primitive data types, String)
b. Reference or Object type properties
c. Collection properties (Array, List, Set, Map, Properties, and etc...)

3.**Lookup Method Injection**

In Spring application there are many beans injected to each other for a goal. There is no problem when injected beans have same scope of beans like singleton bean injected with other singleton beans.

Sometimes in Spring, Problems arise when you need to inject a prototype-scoped bean in a singleton-scoped bean.

Since singletons are created (and then injected) during context creation it's the only time the Spring context is accessed and thus prototype-scoped beans are injected only once, thus defeating their purpose.

But Spring provides another way for injection of beans, It is called method injection. It is solution of above problem in injecting different scoped beans. It works as that since

singleton beans are instantiated at context creation, but it changes the way prototype-scoped are handled, from injection to created by an abstract method. It is actually a another kind of injection used for dynamically overriding a class and its abstract methods to create instances every time the bean is injected.

Method injection is different from Constructor Injection and Setter Injection. While in Constructor and Setter Injection, Spring creates the beans and injects them using the constructor or setter method, in Method Injection Spring overrides a given abstract method of an abstract class and provides an implementation of the overridden method. Note there is an alternative to method injection would be to explicitly access the Spring context to get the bean yourself. It's a bad thing to do since it completely defeats the whole Inversion of Control pattern, but it works.

This is an advanced form of dependency injection and should be used in very special cases as it involves byte-code manipulation by Spring.

Let's consider an example

XML Based Configuration

```java
package com.praveen.spring.core.di.lookupmethod;

import org.springframework.context.annotation.Scope;
import org.springframework.stereotype.Component;

@Component
@Scope("prototype")
public class Token {

}
```

```java
package com.praveen.spring.core.di.lookupmethod;

public abstract class TokenMachine {

    public void findToken() {
        System.out.println("Token has been generated " + generateToken());
    }

    public abstract Token generateToken();

    private Token token;

    public Token getToken() {
        return token;
    }

    public void setToken(Token token) {
        this.token = token;
    }

}
```

```
| beanslookupmethod.xml ⊠ |

 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
 3  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
 4  <beans>
 5  <bean id="token" class="com.praveen.spring.core.di.lookupmethod.Token" scope="prototype" />
 6  <bean id="tokenMachine" class="com.praveen.spring.core.di.lookupmethod.TokenMachine"
 7  scope="singleton">
 8  <lookup-method bean="token" name="generateToken" />
 9  </bean>
L0  </beans>
```

```java
package com.praveen.spring.core.di.lookupmethod;

import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class XMLTestMain {
    public static void main(String[] args) {
        AbstractApplicationContext context = new ClassPathXmlApplicationContext("beanslookupmethod.xml");
        TokenMachine machine = (TokenMachine) context.getBean("tokenMachine");
        machine.findToken();
        machine = (TokenMachine) context.getBean("tokenMachine");
        machine.findToken();
        context.close();
        }

}
```

Output

```
Oct 08, 2019 4:41:57 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@14514713: startup date [Tue Oct 08 16:41:57 IST 2019]; root of context hierarchy
Oct 08, 2019 4:41:57 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [beanslookupmethod.xml]
Oct 08, 2019 4:41:58 PM org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@14514713: startup date [Tue Oct 08 16:41:57 IST 2019]; root of context hierarchy
Token has been generated com.praveen.spring.core.di.lookupmethod.Token@4e7dc304
Token has been generated com.praveen.spring.core.di.lookupmethod.Token@64729b1e
```

If you go with Annotation based configuration then you just need to replace the beans.xml with AppConfig.java

```java
package com.praveen.spring.core.di.lookupmethod;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class AppConfig {
@Bean
public TokenMachine tokenMachine(){
  return new TokenMachine(){

    @Override
    public Token generateToken() {
      return new Token();
    }

  };
 }
}
```

```
package com.praveen.spring.core.di.lookupmethod;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class AnnotationTestMain {
    public static void main(String[] args) {

        AnnotationConfigApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        TokenMachine machine = context.getBean(TokenMachine.class);
        machine.findToken();
        machine = context.getBean(TokenMachine.class);
        machine.findToken();
        context.close();
    }
}
```

Output

```
Oct 08, 2019 4:34:15 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb0d902: startup date [Tue Oct 08 16:34:15 IST 2019]; root of context hierarch
Token has been generated com.praveen.spring.core.di.lookupmethod.Token@4e41089d
Token has been generated com.praveen.spring.core.di.lookupmethod.Token@32a068d1
Oct 08, 2019 4:34:15 PM org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb0d902: startup date [Tue Oct 08 16:34:15 IST 2019]; root of context hierarchy
```

Spring Circular Dependency

If two classes are dependent on each other, then creating object will be very difficult. In this case spring provides best solution.

First it creates objects using default constructor then it performs injections.

For Example:

Consider Employee and Address are the two classes which are having dependency on each other.

```
package com.praveen.spring.core.di.circular.bean;

public class Employee {
    private Address addr;

    public Employee() {
        super();
        System.out.println("In Employee Default Constructor");
    }

    public Address getAddr() {
        return addr;
    }

    public void setAddr(Address addr) {
        this.addr = addr;
        System.out.println("In Employee class , Address setter");
    }
}
```

```java
package com.praveen.spring.core.di.circular.bean;

public class Address {
    private Employee emp;

    public Address() {
        super();
        System.out.println("In Address Default Constructor");
    }

    public Employee getEmp() {
        return emp;
    }

    public void setEmp(Employee emp) {
        this.emp = emp;
        System.out.println("In Address class , Employee setter");
    }

}
```

🗙 beanscircular.xml ⬚

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <bean class="com.praveen.spring.core.di.circular.bean.Employee"
        name="empObjCircular">
        <property name="addr">
            <ref bean="addrObjCircular" />
        </property>
    </bean>
    <bean class="com.praveen.spring.core.di.circular.bean.Address"
        name="addrObjCircular">
        <property name="emp">
            <ref bean="empObjCircular" />
        </property>
    </bean>
</beans>
```

```java
package com.praveen.spring.core.di.circular.test;

import org.springframework.context.ApplicationContext;

public class EmployeeTest {
    private static ApplicationContext context;

    public static void main(String[] args) throws Exception {

        context = new ClassPathXmlApplicationContext("beanscircular.xml");
        Employee cons = context.getBean("empObjCircular", Employee.class);
        System.out.println(cons);
    }

}
```

Output

```
Oct 08, 2019 4:50:03 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@14514713: startup date [Tue Oct 08 16:50:03 IST 2019]; root of context hierarchy
Oct 08, 2019 4:50:03 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [beanscircular.xml]
In Employee Default Constructor
In Address Default Constructor
In Address class , Employee setter
In Employee class , Address setter
com.praveen.spring.core.di.circular.bean.Employee@39c0f4a
```

Let's discuss Circular dependencies using Annotations

Circular dependencies is the scenario when two or more beans try to inject each other via constructor.

It's not possible to write a compilable code which can initialize exactly one instance of each A and B and pass to each other's constructor. We can, however, refactor our above code and can pass circular references via setters but that would kill the semantics of 'initializing mandatory final fields via constructors.

The same problem Spring faces when it has to inject the circular dependencies via constructor. Spring throws BeanCurrentlyInCreationException in that situation.

Example

```java
package com.praveen.spring.core.di.circular.annotations;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.stereotype.Component;

@ComponentScan(basePackageClasses = CircularDependencyExample.class, useDefaultFilters = false,
//scan only the nested beans of this class
        includeFilters = { @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, value = {
                CircularDependencyExample.BeanB.class, CircularDependencyExample.BeanA.class }) })
@Configuration
public class CircularDependencyExample {

    private static ApplicationContext context;

    public static void main(String[] args) {
        context = new AnnotationConfigApplicationContext(CircularDependencyExample.class);
    }

    @Component
    public static class BeanA {
        private final BeanB beanB;

        public BeanA(BeanB b) {
            this.beanB = b;
        }
    }

    @Component
    public static class BeanB {
        private final BeanA beanA;
```

```
    @Component
    public static class BeanB {
        private final BeanA beanA;

        public BeanB(BeanA beanA) {
            this.beanA = beanA;
        }
    }
}
```

Output

Sep 08, 2019 1:50:37 AM
org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing
org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb0d90
2: startup date [Sun Sep 08 01:50:37 IST 2019]; root of context hierarchy
Sep 08, 2019 1:50:38 AM
org.springframework.context.support.AbstractApplicationContext refresh
WARNING: Exception encountered during context initialization - cancelling refresh
attempt: org.springframework.beans.factory.UnsatisfiedDependencyException:

Fixing circular dependencies by using setter injection

```
package com.praveen.spring.core.di.circular.annotations;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.stereotype.Component;

@ComponentScan(basePackageClasses = CircularDependencyExampleUsingSetter.class, useDefaultFilters = false, includeFilters = {
        @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, value = {
                CircularDependencyExampleUsingSetter.BeanB.class, CircularDependencyExampleUsingSetter.BeanA.class }) })
@Configuration
public class CircularDependencyExampleUsingSetter {

    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(CircularDependencyExampleUsingSetter.class);
        BeanA beanA = context.getBean(BeanA.class);
        beanA.doSomething();
    }

    @Component
    static class BeanA {
        private BeanB beanB;

        public BeanA() {
        }

        public void setB(BeanB b) {
            this.beanB = b;
        }

        public void doSomething() {
            System.out.println("doing something");
        }
```

```
        }
    @Component
    static class BeanB {
        private BeanA beanA;

        public BeanB() {
        }

        public void setBeanA(BeanA beanA) {
            this.beanA = beanA;
        }
    }
}
```

Output

```
Oct 08, 2019 4:56:51 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb0d9d2: startup date [Tue Oct 08 16:56:51 IST 2019]; root of context hierarch
doing something
```

Fixing circular dependencies by using @Lazy at constructor injection point

```
package com.praveen.spring.core.di.circular.annotations;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.FilterType;
import org.springframework.context.annotation.Lazy;
import org.springframework.stereotype.Component;

@ComponentScan(basePackageClasses = CircularDependencyExampleUsingLazy.class, useDefaultFilters = false, includeFilters = {
        @ComponentScan.Filter(type = FilterType.ASSIGNABLE_TYPE, value = {
                CircularDependencyExampleUsingLazy.BeanB.class, CircularDependencyExampleUsingLazy.BeanA.class }) })
@Configuration
public class CircularDependencyExampleUsingLazy {

    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(CircularDependencyExampleUsingLazy.class);
        BeanA beanA = context.getBean(BeanA.class);
        beanA.doSomething();
    }

    @Component
    static class BeanA {
        private final BeanB beanB;

        BeanA(@Lazy BeanB b) {
            this.beanB = b;
        }

        public void doSomething() {
            beanB.doSomething();
        }
    }
```

```
        }

    @Component
    static class BeanB {
        private final BeanA beanA;

        BeanB(BeanA beanA) {
            this.beanA = beanA;
        }

        public void doSomething() {
            System.out.println("doing something");
        }
    }
}
```

Output

```
Oct 08, 2019 4:58:46 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh

INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@6c20d902: startup date [Tue Oct 08 16:58:46 IST 2019]; root of context hierarch

doing something
```

Spring Autowiring

Autowiring is a concept of injecting a spring bean automatically without writing <ref/> tag by programmer. Programmer does not required to write explicitly Dependency Injection.

Autowiring can be configured in 2 ways:

1) XML based
2) Annotation Based

@Autowired by default operate byType and if its type is not available then it checks the name.If name and type not available then it checks if there are any qualifier for that bean using annotation @Qualifier("praveen")

All these processing will be done by BeanPostProcessor for each annotation.

XML Based

XML based autowiring can be classified as

a) **byName** : compares the spring bean (java code) filed name (variable name) and configuration code (XML file) bean name (bean tag name) , if they are matched then automatically those objects will be bonded with each other using setter injection.
b) **byType**: It compares bean class variable Data type and spring bean class type. If both are matched then it will do setter injection.
c) **constructor**: It check for parameterized constructor for creating object with reference type as parameter. If not found then uses default constructor at last. Always checks More number of parameters first, if not matched then next level less no of parameters constructor will be compared, and then goes on up to default constructor.
d) **no** : it will not do any auto wiring. It is disabled by default.
e) **autodetect** (works only in older versions like 2.X) : It works like byType if default constructor is available in spring bean, if not works like constructor if parameterized constructor is available.

Let's consider the below example for above concept. Employee class has a Address class Dependency.

```java
package com.praveen.spring.core.autowire.bean;

public class Address {
    private int addrId;
    private String loc;

    public int getAddrId() {
        return addrId;
    }

    public void setAddrId(int addrId) {
        this.addrId = addrId;
    }

    public String getLoc() {
        return loc;
    }

    public void setLoc(String loc) {
        this.loc = loc;
    }

    @Override
    public String toString() {
        return "Address [addrId=" + addrId + ", loc=" + loc + "]";
    }

}
```

```java
package com.praveen.spring.core.autowire.bean;

public class Employee {
    private Address address;

    public Employee() {
        super();
        System.out.println("In default");
    }

    public Employee(Address address) {
        super();
        this.address = address;
        System.out.println("in Param");
    }

    public Address getAddress() {
        return address;
    }

    public void setAddress(Address address) {
        this.address = address;
    }

    @Override
    public String toString() {
        return "Employee [addr=" + address + "]";
    }

}
```

```
package com.praveen.spring.core.autowire.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.praveen.spring.core.autowire.bean.Employee;

public class EmployeeTest {
    private static ApplicationContext context;

    public static void main(String[] args) {
        context = new ClassPathXmlApplicationContext("beansautowire.xml");
        Employee obj = (Employee) context.getBean("empObj");
        System.out.println(obj);

    }

}
```

X beansautowire.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
<bean class="com.praveen.spring.core.autowire.bean.Address" id="address">
<property name="addrId" value="102" />
<property name="loc" value="HYD" />
</bean>
 <bean class="com.praveen.spring.core.autowire.bean.Employee" id="empObj" autowire="byType">
<!--<bean class="com.praveen.spring.core.autowire.bean.Employee" id="employee" autowire="byName"> -->
<!-- <bean class="com.praveen.spring.core.autowire.bean.Employee" id="empObj" autowire="constructor">  -->
</bean>
</beans>
```

Output:

```
Oct 08, 2019 5:06:20 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@14514713; startup date [Tue Oct 08 17:06:20 IST 2019]; root of context hierarchy
Oct 08, 2019 5:06:20 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [beansautowire.xml]
In default
Employee [addr=Address [addrId=102, loc=HYD]]
```

Using Annotation approach

We will use the Employee and Address classes used in xml based approach.

```
package com.praveen.spring.core.autowire.annotations.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.praveen.spring.core.autowire.annotations.bean.Address;
import com.praveen.spring.core.autowire.annotations.bean.Employee;

@Configuration
@ComponentScan(basePackages = "com.praveen.spring.core.autowire.annotations")
public class EmployeeConfig {
    @Bean(name="myEmployee")
    public Employee getBean() {
        Employee emp= new Employee();
        Address addr= new Address();
        addr.setAddrId(102);
        addr.setLoc("HYD");
        emp.setAddress(addr);
        return emp;
    }

}
```

```
package com.praveen.spring.core.autowire.annotations.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.praveen.spring.core.autowire.annotations.bean.Employee;
import com.praveen.spring.core.autowire.annotations.config.EmployeeConfig;

public class EmployeeTest {
    private static ApplicationContext context;

    public static void main(String[] args) {
        context = new AnnotationConfigApplicationContext(EmployeeConfig.class);
        Employee obj = (Employee)context.getBean(Employee.class);
        System.out.println(obj);


    }

}
```

Output

```
Oct 08, 2019 5:19:31 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@5cb0d902: startup date [Tue Oct 08 17:19:31 IST 2019]; root of context hierarch
In default
Employee [addr=Address [addrId=102, loc=HYD]]
```

Spring Collections

Injecting Collection

Tag Name	Inner Tag Name	Java Collection Type	Specification
<list>	<value>	java.util.List<E>	Allows duplicate entries
<map>	<entry>	java.util.Map<K, V>	Key-Value pair of any object type
<set>	<value>	java.util.Set<E>	Does not allow duplicate entries
<props>	<prop>	java.util.Properties	Key-Value pair of type 'String'

In Spring basically collections are configured as
1) List
2) Set
3) Map
4) Properties

To specify above collections , tag are <list>,<set>,<map> and <props>

Here <map> contains internally entries (<entry>). Every entry contains <key> and <value> , these even can be represents as attributes.

Properties also stores data in the key value pair only. But by default key and values are type String in Properties. It contains child tag <prop key=""></prop>. It doesn't contain any <value> tag to represent the value; we can directly specify the value in between the <prop> tag.

For example

```java
package com.praveen.spring.core.collections.bean;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

public class Employee {

    private List<String>empList;
    private Set<String>empSet;
    private Map<String,String>empMap;
    private Properties empProperties;
    public List<String>getEmpList() {
        return empList;
    }
    public void setEmpList(List<String> empList) {
        this.empList = empList;
    }
    public Set<String>getEmpSet() {
        return empSet;
    }
    public void setEmpSet(Set<String> empSet) {
        this.empSet = empSet;
    }
    public Map<String, String>getEmpMap() {
        return empMap;
    }
    public void setEmpMap(Map<String, String> empMap) {
        this.empMap = empMap;
    }
    public Properties getEmpProperties() {
        return empProperties;
    }
```

```xml
beanscollections.xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>
    <bean class="com.praveen.spring.core.collections.bean.Employee"
        name="emp">
        <property name="empList"><!-- it will maintain duplicates -->
            <list>
                <value>A</value>
                <value>A</value>
                <value>A</value>
                <value>B</value>
                <value>C</value>
                <value>D</value>
            </list>
        </property>
        <property name="empSet"><!-- it will not maintain duplicates -->
            <set>
                <value>A</value>
                <value>A</value>
                <value>A</value>
                <value>B</value>
                <value>C</value>
                <value>D</value>
            </set>
        </property>
        <property name="empMap">
            <map>
                <entry>
                    <key>
                        <value>K1</value>
                    </key>
                    <value>V1</value>
                </entry>
```

```xml
            <entry key="K2" value="V2" />
            <entry key="K3">
                <value>V3</value>
            </entry>
            <entry value="V4">
                <key>
                    <value>K4</value>
                </key>
            </entry>
        </map>
    </property>
    <property name="empProperties">
        <props>
            <prop key="key1">
                val1
            </prop>
            <prop key="key1">
                val3
            </prop>
            <prop key="key2">
                val2
            </prop>
        </props>
    </property>
    </bean>
</beans>
```

```java
package com.praveen.spring.core.collections.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.praveen.spring.core.collections.bean.Employee;

public class SpringCollectionsTest {
    private static ApplicationContext context;

    public static void main(String[] args) throws Exception, Exception  {
        context = new ClassPathXmlApplicationContext("beanscollections.xml");
        Employee emp=(Employee)context.getBean("emp");
        System.out.println(emp);

    }


}
```

Output

Oct 08, 2019 5:26:04 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh

INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@145147l3: startup date [Tue Oct 08 17:26:04 IST 2019]; root of context hierarchy

Oct 08, 2019 5:26:04 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions

INFO: Loading XML bean definitions from class path resource [beanscollections.xml]

Employee [empList=[A, A, A, B, C, D], empSet=[A, B, C, D], empMap={K1=V1, K2=V2, K3=V3, K4=V4}, empProperties={key2=val2, key1=val3}]

Note:

1. Employee Object Created with empList as null
2. Employee Object created with empList with zero size (Or empty List)

3. In case of any property tag is defined without any value tag, then Spring Container throws Exception:
org.springframework.beans.factory.parsing.BeanDefinitionParsingException: Configuration problem: <property> element for property 'empList' must specify a ref or value

Using Annotations

```java
package com.praveen.spring.core.collections.annotations.bean;

import java.util.List;

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

public class CollectionsBean {

    @Autowired
    private List<String> nameList;

    public void printNameList() {
        System.out.println(nameList);
    }

}
```

```java
package com.praveen.spring.core.collections.annotations.config;

import java.util.Arrays;
import java.util.List;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.praveen.spring.core.collections.annotations.bean.CollectionsBean;

@Configuration
public class CollectionConfig {

    @Bean
    public CollectionsBean getCollectionsBean() {
        return new CollectionsBean();
    }

    @Bean
    public List<String> nameList() {
        return Arrays.asList("Praveen", "Prasad", "Bhaja");
    }


}
```

```
package com.praveen.spring.core.collections.annotations.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.praveen.spring.core.collections.annotations.bean.CollectionsBean;
import com.praveen.spring.core.collections.annotations.config.CollectionConfig;

public class CollectionsTest {

    private static ApplicationContext context;

    public static void main(String args[]) {
        context = new AnnotationConfigApplicationContext(CollectionConfig.class);
        CollectionsBean collectionsBean = context.getBean(
          CollectionsBean.class);
        collectionsBean.printNameList();
    }

}
```

Output

```
Oct 08, 2019 5:29:17 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.annotation.AnnotationConfigApplicationContext@6c8d902: startup date [Tue Oct 08 17:29:17 IST 2019]; root of context hierarc
[Praveen, Prasad, Bhaja]
```

Spring Interceptor

- ✓ An Interceptor runs between requests whereas a Filter runs before rendering view
- ✓ Filter is related to the Servlet API and HandlerIntercepter is a Spring specific concept.
- ✓ A Servlet Filter is used in the web layer only, you can't use it outside of a web context. Interceptors can be used anywhere which is the main difference.
- ✓ For authentication of web pages you would use a servlet filter.
  For security stuff in your business layer or logging you would use an Interceptor.

Filter

A filter as the name suggests is a Java class executed by the servlet container for each incoming HTTP request and for each http response. This way, is possible to manage HTTP incoming requests before them reach the resource, such as a JSP page, a servlet or a simple static page; in the same way is possible to manage HTTP outbound response after resource execution.

Interceptor

Spring Interceptors are similar to Servlet Filters but they acts in Spring Context so are many powerful to manage HTTP Request and Response but they can implement more sophisticated behavior because can access to all Spring context.

Methods in HandlerInterceptorAdapter

- ✓ prehandle method- do something before hitting the controller.
- ✓ posthandle method- do something after returning from controller.
- ✓ after completion method- do something after response is sent out and the view is rendered.

ResponseEntity is meant to represent the entire HTTP response. You can control anything that goes into it: status code, headers, and body.

Let's see an Example

Once the below code is included we will see the below logs

In RequestHeaderInterceptor :: SPRING Called postHandle method for URI /SpringMVCExample/users/createUser

In RequestHeaderInterceptor :: SPRING Called afterCompletion method for URI /SpringMVCExample/users/createUser

In RequestHeaderInterceptor :: SPRING Called postHandle method for URI /SpringMVCExample/users/getUser

In RequestHeaderInterceptor :: SPRING Called afterCompletion method for URI /SpringMVCExample/users/getUser

```java
package com.praveen.interceptor;

import javax.servlet.http.HttpServletRequest;

@Component
public class RequestHeaderInterceptor extends HandlerInterceptorAdapter {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {

//      if (StringUtils.isBlank(request.getHeader("user-auth-key"))) {
//          throw new InvalidHeaderFieldException("Invalid request");
//      }

        return super.preHandle(request, response, handler);
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
            @Nullable ModelAndView modelAndView) throws Exception {

        System.out.println("In RequestHeaderInterceptor  :: SPRING Called postHandle method for URI "+request.getRequestURI() );
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,
            @Nullable Exception ex) throws Exception {

        System.out.println("In RequestHeaderInterceptor  :: SPRING Called afterCompletion method for URI "+request.getRequestURI() );
    }

}
```

```java
package com.praveen.config;

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

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.praveen"})
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;
    }

    @Autowired
    private RequestHeaderInterceptor requestHeaderInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        // TODO Auto-generated method stub
        registry.addInterceptor(requestHeaderInterceptor);
    }

}
```

Spring Listener

Application Listener can generically declare the event type that it is interested in. When registered with a Spring Application Context, events will be filtered accordingly, with the listener getting invoked for matching event objects only.

Application events are available since the very beginning of the Spring framework as a mean for loosely coupled components to exchange information.

Types of event are available

- ✓ Start – ContextStartedEvent.
- ✓ Stop – ContextStoppedEvent.
- ✓ Refresh – ContextRefreshedEvent.
- ✓ Close – ContextClosedEvent.

```
package com.praveen.listeners;

import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextRefreshedEvent;
import org.springframework.stereotype.Component;

@Component
public class ContextRefreshHandler implements ApplicationListener<ContextRefreshedEvent> {
    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {
        System.out.println("Context Refreshed Event Received " + event.toString());
    }
}
```

Output

Context Refreshed Event Received
org.springframework.context.event.ContextRefreshedEvent[source=WebApplicationCon
text for namespace 'dispatcher-servlet': startup date [Tue Oct 08 23:49:52 IST 2019];
root of context hierarchy]

```
package com.praveen.listeners;

import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextStartedEvent;
import org.springframework.stereotype.Component;

@Component
public class ContextStartHandler implements ApplicationListener<ContextStartedEvent> {
    @Override
    public void onApplicationEvent(ContextStartedEvent event) {
        System.out.println("Context Started Event Received " + event.toString());
    }
}
```

```
package com.praveen.listeners;

import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextStoppedEvent;
import org.springframework.stereotype.Component;

@Component
public class ContextStopHandler implements ApplicationListener<ContextStoppedEvent> {
    @Override
    public void onApplicationEvent(ContextStoppedEvent event) {
        System.out.println("Context Stopped Event Received " + event.toString());
    }
}
```

```java
package com.praveen.listeners;

import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextClosedEvent;
import org.springframework.stereotype.Component;

@Component
public class ContextCloseHandler implements ApplicationListener<ContextClosedEvent> {
    @Override
    public void onApplicationEvent(ContextClosedEvent event) {
        System.out.println("Context Closed Event Received " + event.toString());
    }
}
```

Let's create a customized listener

CreateLogEvent.java

package com.praveen.restservices.handler;

import org.springframework.context.ApplicationEvent;

public class CreateLogEvent extends ApplicationEvent {

private static final long serialVersionUID = 391259937537604095L;

private String message;

public CreateLogEvent(Object source, String message) {
super(source);
this.message = message;
}
public String getMessage() {
return message;
}
}

PraveenLogDAOImpl.java

package com.praveen.restservices.dao.impl;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.List;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;
import org.springframework.jdbc.support.GeneratedKeyHolder;
import org.springframework.jdbc.support.KeyHolder;
import org.springframework.stereotype.Repository;

```java
import com.praveen.restservices.config.PraveenLogRowMapper;
import com.praveen.restservices.dao.PraveenLogDAO;
import com.praveen.restservices.handler.CreateLogEvent;
import com.praveen.restservices.model.PraveenLog;

@Repository
public class PraveenLogDAOImpl implements PraveenLogDAO {

private JdbcTemplate jdbcTemplate;
@Autowired
PraveenLogDAOImpl(DataSource dataSource1) {
this.jdbcTemplate = new JdbcTemplate(dataSource1);
}

@Autowired
ApplicationEventPublisher publisher;

@Override
public PraveenLog createLog(PraveenLog praveenLog) throws Exception {
final String sql = "insert into
PRAVEENLOG(PRAVEENLOG_MESSAGE,PRAVEENLOG_DATE) values(?,?)";
KeyHolder holder = new GeneratedKeyHolder();
jdbcTemplate.update(new PreparedStatementCreator() {
@Override
public PreparedStatement createPreparedStatement(Connection connection) throws
SQLException {
PreparedStatement ps = connection.prepareStatement(sql,
Statement.RETURN_GENERATED_KEYS);
ps.setString(1, praveenLog.getLogMessage());
ps.setDate(2, praveenLog.getLogDate());
return ps;
}
}, holder);
int praveenLogId = holder.getKey().intValue();
System.out.println("Publishing Create Log Event");
praveenLog.setLogId(praveenLogId);
CreateLogEvent cle= new CreateLogEvent(this,"Praveen");
publisher.publishEvent(cle);
return praveenLog;
}

@Override
public List<PraveenLog> getAllLogs() {
return jdbcTemplate.query("select * from PRAVEENLOG", new
PraveenLogRowMapper());
}

}
```

}

CreateLogHandler.java

package com.praveen.restservices.handler;

import org.springframework.context.ApplicationListener;
import org.springframework.stereotype.Component;

@Component
public class CreateLogHandler implements ApplicationListener<CreateLogEvent> {

@Override
public void onApplicationEvent(CreateLogEvent event) {
System.out.println("Received Create Log Event " + event.getMessage());

}

}

Output

Publishing Create Log Event
Received Create Log Event Praveen

Spring AOP

What is AOP(Aspect Oriented Programming)?

AOP is a programming methodology which helps to manage cross cutting concern like transaction management, logging,security etc

Why AOP?

- It provides the pluggable way to dynamically add the additional concern before,after or around the actual business logic.
- Aspects enable modularization of concerns.

Uses of Spring AOP

- ✓ Remove tight coupling between cross cutting concerns and business logic.
- ✓ Easy to maintain code in present and future.

AOP Core Concepts

Aspect

An Aspect is a class that implements the Java Enterprise application concerns which cuts through multiple classes like transaction management,logging,security etc.An aspect will be configured using annotation @Aspect.

To enable AspectJ annotations in Spring projects, we need to configure @EnableAspectJAutoProxy in the aspect class and incase of spring boot project we need to configure @EnableAspectJAutoProxy in main class.

Advice

Advice represents an action taken by an aspect at a particular join point.

1. @Before – Adviced to execute before the joint point.
2. @AfterReturning – Adviced to execute after the joint point complete.
3. @After – Adviced to execute after the joint point.
4. @AfterThrowing- Adviced to execute after the joint point exists by throwing an exception
5. @Around – Advice that surrounds a joint point like method invocation

Joinpoint

A point in a program such as method execution,exception handling etc.

Pointcut

A pointcut is an expression language of AOP that matches joint points which determines whether Advice needs to be executed or not.
Execution
This
Target
Args
@target
@args
@within
@annotation

Advisor

An advisor is a combination of a pointcut and an advice.
An advisor knows about what advices are needed for what joinpoints identified by a pointcut.

Target & Proxy

Target is a business class object, before AOP feature is applied for it.
Proxy is also a business class object, after AOP feature is applied to it.

Weaving

Linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.
Weaving is a process of converting a target object into a proxy object by adding advices

to it.

For example, a web-container converts a jsp into equivalent servlet. Here jsp is a target and servlet is a proxy. This process is called weaving.

Annotations Allowed Attributes

- ✓ @Pointcut value
- ✓ @Before value
- ✓ @AfterReturning pointcut, value, returning
- ✓ @After value
- ✓ @AfterThrowing pointcut, value, throwing
- ✓ @Around value

Dependencies

```xml
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aop</artifactId>
    <version>5.1.1.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjtools</artifactId>
    <version>1.9.2</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>1.9.2</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.9.2</version>
</dependency>
```

Here with the example for @Before,@After,@AfterReturning,@AfterThrowing and @Around.

```java
package com.praveen.spring.aop.service;

import org.springframework.stereotype.Service;

@Service
public class EmployeeService {
    public String addEmployee() {
        System.out.println("Add Employee ");
        return "Employee Praveen information is added successfully";
    }

    public void modifyEmployee() {
        System.out.println("Modify Employee");
    }

    public void deleteEmployee() {
        System.out.println("Delete Employee");
    }

}
```

```java
package com.praveen.spring.aop.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

import com.praveen.spring.aop.service.EmployeeService;


@Configuration
@ComponentScan(basePackages = "com.praveen.spring.aop")
public class EmployeeConfig {
    @Bean(name="employeeService")
    public EmployeeService getBean() {
        return new EmployeeService();
    }

}
```

```java
package com.praveen.spring.aop.aspect;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.stereotype.Component;

@Aspect
@Component
@EnableAspectJAutoProxy
public class LoggingAspect {

    @Before("addEmployee()")
    public void logBefore(JoinPoint joinPoint) {

        System.out.print("logBefore() is running!");
        System.out.println(", before " + joinPoint.getSignature().getName() + " method");
        System.out.println("******");
    }

    @After("addEmployee()")
    public void logAfter(JoinPoint joinPoint) {

        System.out.print("logAfter() is running!");
        System.out.println(", after " + joinPoint.getSignature().getName() + " method");
        System.out.println("******");
    }
```

```java
    @AfterReturning(pointcut = "addEmployee()", returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {

        System.out.print("logAfterReturning() is running!");
        System.out.println(", after " + joinPoint.getSignature().getName() + " method");
        System.out.println("Method returned value is = " + result);
        System.out.println("******");
    }

    @AfterThrowing(pointcut = "addEmployee()", throwing = "exception")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable exception) {

        System.out.print("logAfterThrowing() is running!");
        System.out.println(", after " + joinPoint.getSignature().getName() + " method throwing exception");
        System.out.println("exception = " + exception);
        System.out.println("******");
    }

    @Pointcut("execution(* com.praveen.spring.aop.service.EmployeeService.addEmployee())")
    public void addEmployee() {
    }

    @Around("execution(* com.praveen.spring.aop.service.EmployeeService.modifyEmployee())")
    public void aroundAdvice(ProceedingJoinPoint pjp) throws Throwable {
        System.out.println("Around Advice Initial message");
        pjp.proceed();
        System.out.println("Around Advice later message");

    }

}
```

```java
package com.praveen.spring.aop.test;

import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.AnnotationConfigApplicationContext;

import com.praveen.spring.aop.config.EmployeeConfig;
import com.praveen.spring.aop.service.EmployeeService;

public class EmployeeTest {

    private static ApplicationContext context;

    public static void main(String args[]) {

        context = new AnnotationConfigApplicationContext(EmployeeConfig.class);

        System.out.println("-------------------------------------");

        EmployeeService employeeService = context.getBean(EmployeeService.class);

        employeeService.addEmployee();

        employeeService.modifyEmployee();

        employeeService.deleteEmployee();
    }

}
```

Output

```
----------------------------------------
logBefore() is running!, before addEmployee method
******
Add Employee
logAfter() is running!, after addEmployee method
******
logAfterReturning() is running!, after addEmployee method
Method returned value is = Employee Praveen information is added successfully
******
Around Advice Initial message
Modify Employee
Around Advice later message
Delete Employee
```

Example for @Around for ExecutionTimeTrackerAdvice in Spring MVC project

```java
package com.praveen.aop.advice;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TrackExecutionTime {

}
```

```java
package com.praveen.aop.advice;

import org.apache.log4j.Logger;
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
import org.springframework.context.annotation.EnableAspectJAutoProxy;
import org.springframework.stereotype.Component;

@EnableAspectJAutoProxy
@Aspect
@Component
public class ExecutionTimeTrackerAdvice {

    Logger logger = Logger.getLogger(ExecutionTimeTrackerAdvice.class);

    @Around("@annotation(com.praveen.aop.advice.TrackExecutionTime)")
    public Object trackTime(ProceedingJoinPoint pjp) throws Throwable {
        long stratTime = System.currentTimeMillis();
        Object obj = pjp.proceed();
        long endTime = System.currentTimeMillis();
        logger.info("Method name" + pjp.getSignature() + " time taken to execute : " + (endTime - stratTime));
        return obj;
    }

}
```

Just use the @TrackExecutionTime where you want for example if I include it in one of the controller.

```java
@Controller
public class AddController {

    @GetMapping("/index")
    @TrackExecutionTime
    public String homeInit(Locale locale, Model model) {
        return "index";
    }
}
```

Output

2019-10-08 23:14:32 INFO ExecutionTimeTrackerAdvice:22 - Method nameString com.praveen.controller.AddController.homeInit(Locale,Model) time taken to execute : 36

Let's see another controller where we use @TrackExecutionTime

```java
package com.praveen.bean;

public class User {

    private String password;
    private String userName;
    private int ssoId;

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public int getSsoId() {
        return ssoId;
    }

    public void setSsoId(int ssoId) {
        this.ssoId = ssoId;
    }

}
```

```java
package com.praveen.dao;

import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Repository;

import com.praveen.bean.User;

@Repository
public class UserDAO {

    private List<User> userList;

    public UserDAO() {
        userList = new ArrayList<User>();
    }

    public List<User> getAllUsers() {
        return userList;
    }

    public User createUser(User user) {
        userList.add(user);
        return user;

    }

}
```

```java
package com.praveen.service;

import java.util.List;

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

import com.praveen.bean.User;
import com.praveen.dao.UserDAO;

@Service
public class UserService {

    @Autowired
    private UserDAO userDao;

    public List<User> getUsers() {
        return userDao.getAllUsers();
    }

    public User createUser(User user) {
        return userDao.createUser(user);
    }
}
```

```java
package com.praveen.controller;

import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {

    @Autowired
    private UserService userService;


    @GetMapping
    @RequestMapping("/getUser")
    @TrackExecutionTime
    public List<User> getUsers() {
        return userService.getUsers();
    }

    @PostMapping(value ="/createUser", consumes = MediaType.APPLICATION_JSON_VALUE,
            produces = MediaType.APPLICATION_JSON_VALUE)
    @TrackExecutionTime
    public User createUser(@RequestBody User user) {
        return userService.createUser(user);
    }


}
```

Now perform POST request as shown below in postman as mentioned in screenshot 2

Now perform GET request as shown below in postman as mentioned in screenshot3

Now check the console for the ExecutionTimeTrackerAdvice AOP statements

2019-10-08 23:37:01 INFO ExecutionTimeTrackerAdvice:22 - Method nameUser com.praveen.controller.UserController.createUser(User) time taken to execute : 23

2019-10-08 23:37:07 INFO ExecutionTimeTrackerAdvice:22 - Method nameList com.praveen.controller.UserController.getUsers() time taken to execute : 0
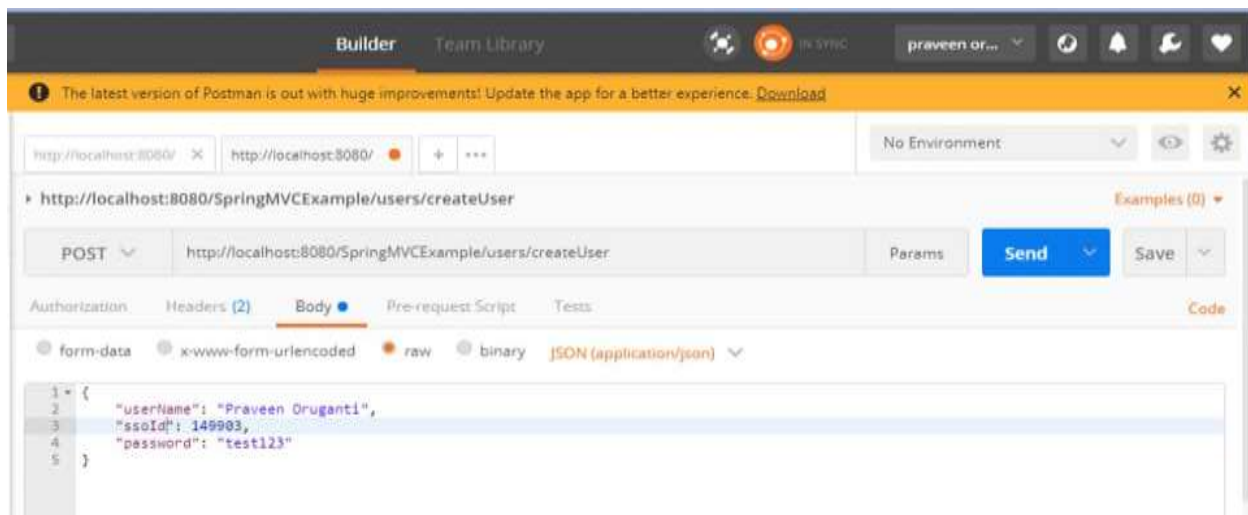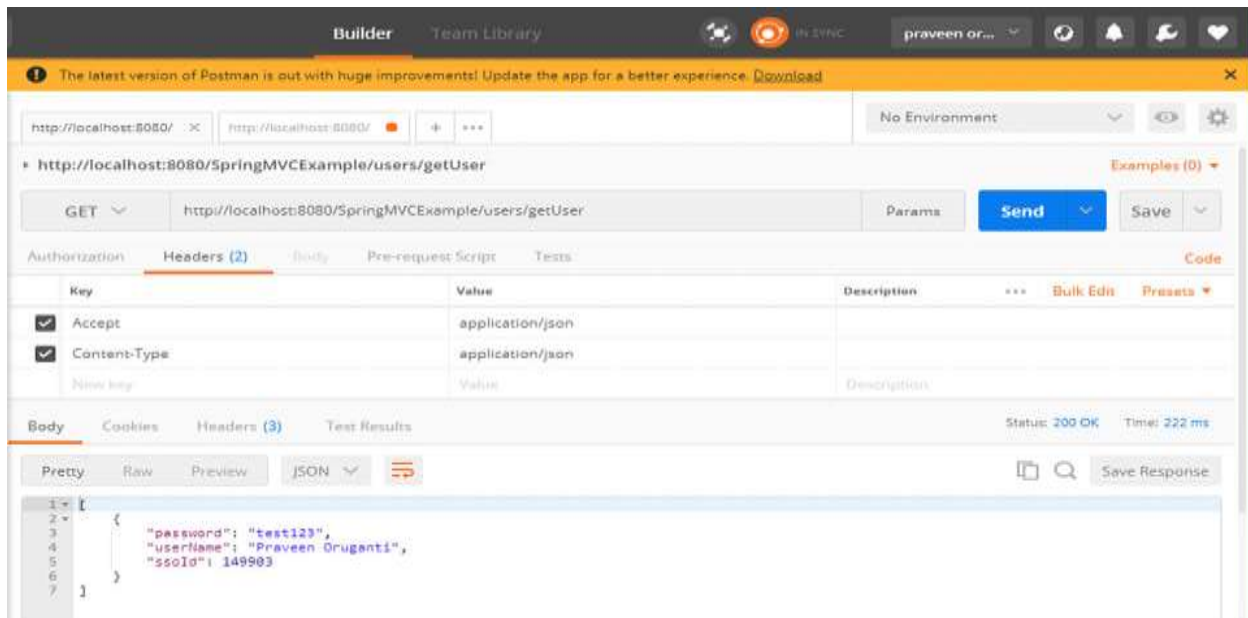
Design patterns used in Spring (Adapter pattern: Spring Security)

Singleton Pattern	: Singleton-scoped beans
Factory Pattern	: Bean Factory classes
Prototype Pattern	: Prototype-scoped beans
Adapter Pattern	: Spring Web and Spring MVC
Proxy Pattern	: Spring Aspect Oriented Programming support
Template Method Pattern	: JdbcTemplate, HibernateTemplate, etc.
Front Controller	: Spring MVC DispatcherServlet
Data Access Object	: Spring DAO support
Model View Controller	: Spring MVC

How a Servlet Application works

Web container is responsible for managing execution of servlets and JSP pages for Java EE application.

When a request comes in for a servlet, the server hands the request to the Web Container. Web Container is responsible for instantiating the servlet or creating a new thread to handle the request. Its the job of Web Container to get the request and response to the servlet. The container creates multiple threads to process multiple requests to a single servlet.

Servlets don't have a main() method. Web Container manages the life cycle of a Servlet instance.

Let's see how it works internally

✓ User sends request for a servlet by clicking a link that has URL to a servlet.

✓ The container finds the servlet using deployment descriptor and creates two objects :
HttpServletRequest
HttpServletResponse

✓ Then the container creates or allocates a thread for that request and calls the Servlet's service() method and passes the request, response objects as arguments.

✓ The service() method, then decides which servlet method, doGet() or doPost() to call, based on HTTP Request Method(Get, Post etc) sent by the client. Suppose the client sent an HTTP GET request, so the service() will call Servlet's doGet() method.

✓ Then the Servlet uses response object to write the response back to the client.

✓ After the service() method is completed the thread dies. And the request and response objects are ready for garbage collection.

Spring MVC

Spring provides Model-View-Controller (MVC) architecture, and components that can be used to develop flexible and loosely coupled web applications. It uses the features of Spring core features like IoC and DI.

1. The Model encapsulates the application data and in general they will consist of POJOs.
2. The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
3. The Controller is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.
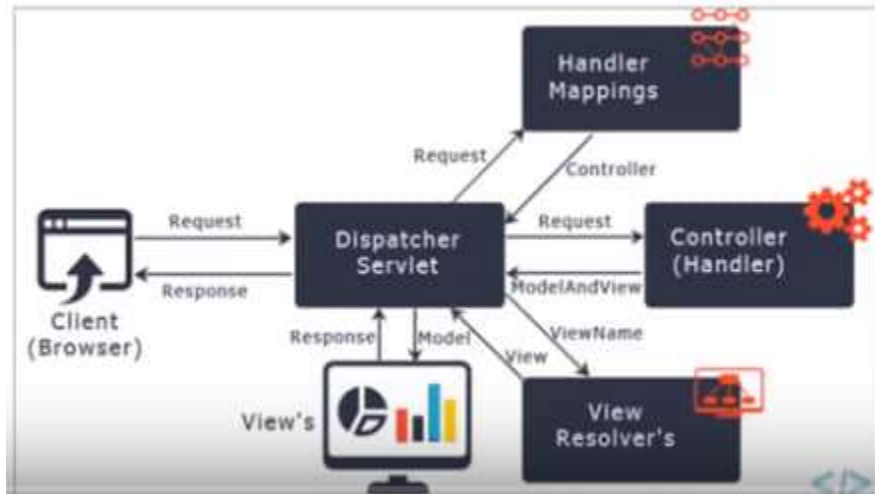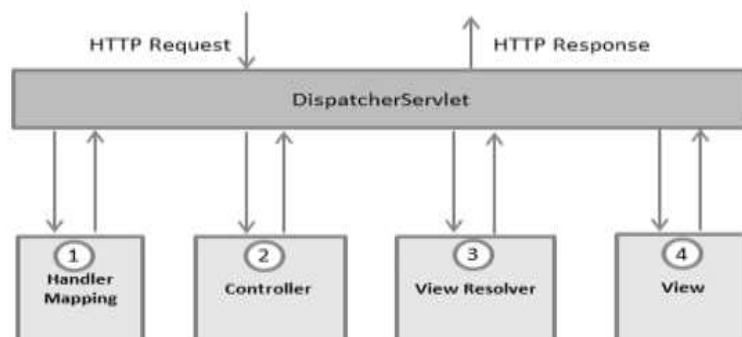
The Workflow

The Spring Web MVC framework is designed around the DispatcherServlet.

Java Servlet are programs that act as a middle layer between a request coming from a Web browser or HTTP client, and applications on the HTTP server.

What happens behind the scene when an HTTP request is sent to the server

The DispatcherServlet handles all the HTTP requests and responses.
After receiving an HTTP request, the DispatcherServlet consults the HandlerMapping to call the appropriate Controller.
The Controller takes the request, calls the appropriate method, and builds the model and returns the view page name to the DispatcherServlet.
The DispatcherServlet will then take help from the ViewResolver to pick up the defined view page for the request.
Once the view is finalized, the DispatcherServlet passes the model data to the view, which is finally rendered on the browser.

All the above-mentioned components (i.e. HandlerMapping, Controller, and ViewResolver) are parts of WebApplicationContext, which is an extension of the ApplicationContext with some extra features necessary for web applications

Spring MVC Application Components

A collection of web pages to layout UI components.
A collections of Spring beans (controllers, services, etc).
Spring configurations (XML, Annotation, or Java).

DispatcherServlet

The DispatcherServlet is responsible for loading web component specific beans like controllers, view resolvers and handler mappings.

DispatcherServlet acts as front controller for Spring based web applications. It provides a mechanism for request processing where actual work is performed by configurable, delegate components. It is inherited from javax.servlet.http.HttpServlet, it is typically configured in the web.xml file.

A web application can define any number of DispatcherServlet instances. Each servlet will operate in its own namespace, loading its own application context with mappings, handlers, etc. Only the root application context as loaded by ContextLoaderListener, if any, will be shared. In most cases, applications have only single DispatcherServlet with the context-root URL(/), that is, all requests coming to that domain will be handled by it.

DispatcherServlet uses Spring configuration classes to discover the delegate components it needs for request mapping, view resolution, exception handling etc.
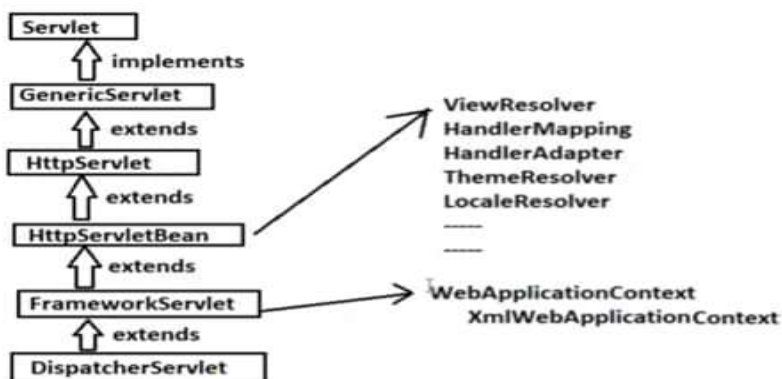
Let's understand how dispatcher servlet works internally?

In a Spring-based application, our application objects live within an object container. This container creates objects and associations between objects, and manages their complete life cycle. These container objects are called Spring-managed beans (or simply beans), and the container is called an application context (via class ApplicationContext) in the Spring world.

WebApplicationContext is an extension of a plain ApplicationContext. it is web aware ApplicationContext i.e it has Servlet Context information. When DispatcherServlet is loaded, it looks for the bean configuration file of WebApplicationContext and initializes it.

By having access to Servlet context, any spring bean which implement ServletConextAware interface can get access to ServletContext instance and do many things with it. For example, it can get context init parameters, get context root information and get resources location inside web application folders.

ContextLoadListener

ContextLoadListener is a Servlet listener that loads all different configuration files(service layer configuration, persistence layer configuration etc) into single spring application context.

This helps to split spring configuration across multiple xml files.

ContextLoadListener creates the root application context and will be shared with child contexts created by all Dispatcheservlet contexts.

web.xml

```
<listener>
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>
/WEB-INF/config/applicationContext-service.xml
/WEB-INF/config/applicationContext-dao.xml
</param-value>
</context-param>
```

Difference between Model, ModelMap and ModelAndView

Model is an interface where as **ModelMap** is a class

ModelAndView is just a container for both ModelMap and a View object. It allows a controller to return both as a single value.

For example

```
@RequestMapping(value = "/process", params = "add")
@TrackExecutionTime
public ModelAndView add(@RequestParam("num1") int num1, @RequestParam("num2") int num2) {
    int result = num1 + num2;
    ModelAndView mv = new ModelAndView();
    mv.setViewName("display");
    mv.addObject("result", result);
    return mv;
}
```

Please note Model objects are specific to request and they cannot be shared across requests.

For example

```
@RequestMapping(value = "/", method = RequestMethod.GET)
public String welcome(Model model){
model.addAttribute("message", "Hello Spring MVC Framework!");
return "welcome-page";

}
```

Can we return response from controller method directly?

The default return value from spring controller is ModelAndView object or logical view name.
We can return response also from controller method using @ResponseBody annotation.

For example

```java
@RequestMapping(value = "/process", params = "duplicate")
@TrackExecutionTime
public @ResponseBody ModelAndView validateNumbers(@RequestParam("num1") int num1, @RequestParam("num2") int num2) {
    ModelAndView mv = new ModelAndView();
    if (num1 == num2) {
        mv.setViewName("error");
    } else {
        int result = num1 + num2;
        mv.setViewName("display");
        mv.addObject("result", result);
    }
    return mv;
}
```

RequestParam

RequestParam is used to read the query parameter

For example

```java
@RequestMapping(value = "/process", params = "add")
@TrackExecutionTime
public ModelAndView add(@RequestParam("num1") int num1, @RequestParam("num2") int num2) {
    int result = num1 + num2;
    ModelAndView mv = new ModelAndView();
    mv.setViewName("display");
    mv.addObject("result", result);
    return mv;
}
```

PathVariable
PathVariable is used to read the path parameter.

For example

```java
@GetMapping("/getBook/{id}")
public @ResponseBody Book getBookDataById(@PathVariable("id") String bookId){
// logic to find book
return book;
}
```

Spring MVC Example using XML configuration

web.xml

```xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xmlns="http://java.sun.com/xml/ns/javaee"
4      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
5      id="WebApp_ID" version="3.0">
6      <display-name>spring-mvc-demo</display-name>
7      <!-- Spring MVC Configs -->
8      <!-- Step 1: Configure Spring MVC Dispatcher Servlet -->
9      <servlet>
10         <servlet-name>MyDispatcher</servlet-name>
11         <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
12         <init-param>
13             <param-name>contextConfigLocation</param-name>
14             <param-value>/WEB-INF/spring-mvc-demo-servlet.xml</param-value>
15         </init-param>
16         <load-on-startup>1</load-on-startup>
17     </servlet>
18     <!-- Step 2: Set up URL mapping of Spring MVC Dispatcher Servlet -->
19     <servlet-mapping>
20         <servlet-name>MyDispatcher</servlet-name>
21         <url-pattern>/</url-pattern>
22     </servlet-mapping>
23 </web-app>
```

spring-mvc-demo-servlet.xml

```xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <beans xmlns="http://www.springframework.org/schema/beans"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xmlns:context="http://www.springframework.org/schema/context"
5      xmlns:mvc="http://www.springframework.org/schema/mvc"
6      xsi:schemaLocation="
7          http://www.springframework.org/schema/beans
8          http://www.springframework.org/schema/beans/spring-beans.xsd
9          http://www.springframework.org/schema/context
10         http://www.springframework.org/schema/context/spring-context.xsd
11         http://www.springframework.org/schema/mvc
12         http://www.springframework.org/schema/mvc/spring-mvc.xsd">
13
14     <!-- Step 3: Add support for component scanning -->
15     <context:component-scan
16         base-package="com.praveen" />
17     <!-- Step 4: Add support for conversion, formatting and validation support -->
18     <mvc:annotation-driven />
19     <!-- Step 5: Define Spring MVC view resolver -->
20     <bean
21         class="org.springframework.web.servlet.view.InternalResourceViewResolver">
22         <property name="prefix" value="/WEB-INF/view/" />
23         <property name="suffix" value=".jsp" />
24
25     </bean>
26 </beans>
```

```java
package com.praveen.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/welcome")
public class WelcomeController {
@RequestMapping(value = "/", method = RequestMethod.GET)
public String welcome(Model model){
model.addAttribute("message", "Hello Spring MVC Framework!");
return "welcome-page";
}
}
```

📄 welcome-page.jsp ⌗

```jsp
1  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2      pageEncoding="ISO-8859-1"%>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <title>Spring MVC Demo</title>
7  </head>
8  <body>
```

Browser: http://localhost:8080/SpringMVCProject/welcome/

http://localhost:8080/SpringMVCProject/welcome/

Hello Spring MVC Framework!

Let's see another example

```
package com.praveen.bean;

public class User1 {
    private String name;
    private Integer id;

    public User1() {
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public Integer getId() {
        return id;
    }
}
```

```java
package com.praveen.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.praveen.bean.User1;

@Controller
public class User1Controller {
    @RequestMapping("/user1")
    public String newUser(Model model) {
        model.addAttribute("user1", new User1());
        return "user1Form";
    }

    @RequestMapping(value = "/addUser1", method = RequestMethod.POST)
    public String addUser(@ModelAttribute("user1") User1 user) {
        return "result";
    }
}
```

result.jsp ⊠

```jsp
 1  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
 2      pageEncoding="ISO-8859-1"%>
 3  <!DOCTYPE html>
 4  <html>
 5  <head>
 6  <title>Spring MVC Demo</title>
 7  </head>
 8  <body>
 9  <h2>User Data</h2>
10  <table>
11  <tr>
12  <td>Name</td>
13  <td>${user1.name}</td>
14  </tr>
15  <tr>
16  <td>ID</td>
17  <td>${user1.id}</td>
18  </tr>
19  </table>
20  </body>
21  </html>
```

user1Form.jsp ⊠

```jsp
 1  <%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
 2  <html>
 3  <head>
 4  <title>Spring MVC Demo</title>
 5  </head>
 6  <body>
 7  <h2>User1 Data</h2>
 8  <form:form action="addUser1" modelAttribute="user1" method="POST">
 9  <form:label path="name">Name</form:label>
10  <form:input path="name" /> <br><br>
11  <form:label path="id">Id</form:label>
12  <form:input path="id" />
13  <input type="submit" value="Submit"/>
14  </form:form>
15  </body>
16  </html>
```
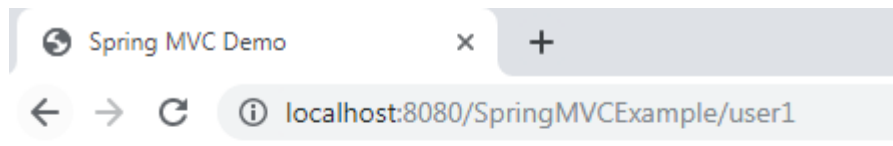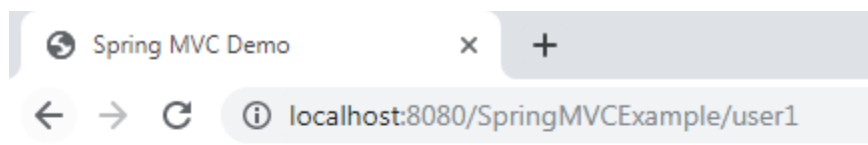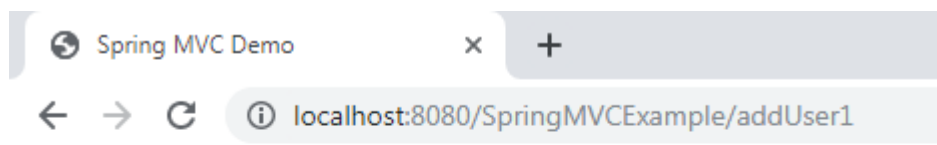
Browser: http://localhost:8080/SpringMVCProject/user1

Spring MVC using annotations

```
📄 AppInitializer.java ⊠

 1  package com.praveen.config;
 2
 3  import org.springframework.web.servlet.support.AbstractAnnotationConfigDispatcherServletInitializer;
 4
 5  public class AppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
 6
 7⊖     @Override
 8      protected Class<?>[] getRootConfigClasses() {
 9          return new Class[] { };
10      }
11
12⊖     @Override
13      protected Class<?>[] getServletConfigClasses() {
14          return new Class[] { WebMvcConfig.class };
15      }
16
17⊖     @Override
18      protected String[] getServletMappings() {
19          return new String[] { "/" };
20      }
21  }
```

```
package com.praveen.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.EnableWebMvc;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
import org.springframework.web.servlet.view.InternalResourceViewResolver;
import org.springframework.web.servlet.view.JstlView;

@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "com.praveen"})
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;
    }
}
```

```java
package com.praveen.controller;

import java.util.Locale;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HomeController {
    @GetMapping("/")
    public String homeInit(Locale locale, Model model) {
        return "home";
    }

    @RequestMapping(value = "/returnHelloWorld", method = RequestMethod.GET)
    @ResponseBody
    public String returnHelloMethod() {
        return "Hello world!";
    }

}
```

📄 home.jsp ⊠

```html
1 <html>
2 <head><title>Home</title></head>
3 <body>
4 <h1>Spring MVC Using Annotation </h1>
5 <h2>Welcome to Praveen Oruganti Homepage</h2>
6 </body>
7 </html>
```

```java
package com.praveen.controller;

import java.util.Locale;

@Controller
public class AddController {

    @GetMapping("/index")
    @TrackExecutionTime
    public String homeInit(Locale locale, Model model) {
        return "index";
    }

    @RequestMapping(value = "/process", params = "add")
    @TrackExecutionTime
    public ModelAndView add(@RequestParam("num1") int num1, @RequestParam("num2") int num2) {
        int result = num1 + num2;
        ModelAndView mv = new ModelAndView();
        mv.setViewName("display");
        mv.addObject("result", result);
        return mv;
    }

    @RequestMapping(value = "/process", params = "duplicate")
    @TrackExecutionTime
    public @ResponseBody ModelAndView validateNumbers(@RequestParam("num1") int num1, @RequestParam("num2") int num2) {
        ModelAndView mv = new ModelAndView();
        if (num1 == num2) {
            mv.setViewName("error");
        } else {
            int result = num1 + num2;
            mv.setViewName("display");
            mv.addObject("result", result);
        }
        return mv;
    }
```

index.jsp

```jsp
1  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2      pageEncoding="ISO-8859-1"%>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="ISO-8859-1">
7  <title>Insert title here</title>
8  </head>
9  <body>
10     <form action="process">
11         <input type="text" name="num1" /><br />
12         <input type="text" name="num2" /><br />
13         <input type = "submit" name = "add" value="add"/>
14         <input type = "submit" name = "duplicate" value="duplicatecheck"/>
15     </form>
16  </body>
17  </html>
```

display.jsp

```jsp
1  <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
2      pageEncoding="ISO-8859-1"%>
3  <!DOCTYPE html>
4  <html>
5  <head>
6  <meta charset="ISO-8859-1">
7  <title>Insert title here</title>
8  </head>
9  <body>
10  Result is ${result}
11  </body>
12  </html>
```

error.jsp

```jsp
1  <html>
2  <head><title>Error</title></head>
3  <body>
4  <h1>Both Numbers are same</h1>
5  </body>
6  </html>
```

Junit Test

```java
package com.praveen.test;

import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.post;

public class AddControllerTest {

    private MockMvc mockMvc;

    private AddController addController;

    @Before
    public void setup(){
        addController = new AddController();
        mockMvc = MockMvcBuilders.standaloneSetup(addController).build();
    }

    @Test
    public void testAdd() throws Exception{
        mockMvc.perform(post("/process").param("num1", "10").param("num2","20").param("add", ""))
            .andExpect(status().isOk())
            .andExpect(view().name("display"));
    }

    @Test
    public void testDuplicate() throws Exception{
        mockMvc.perform(post("/process").param("num1", "20").param("num2","20").param("duplicate", ""))
            .andExpect(status().isOk())
            .andExpect(view().name("error"));
    }

    @Test
    public void testNoDuplicate() throws Exception{
        mockMvc.perform(post("/process").param("num1", "20").param("num2","30").param("duplicate", ""))
            .andExpect(status().isOk())
            .andExpect(view().name("display"));
    }

}
```

Spring MVC Annotations

- ✓ @Controller
- ✓ @RequestMapping
- ✓ @PathVariable
- ✓ @RequestParam
- ✓ @ModelAttribute
- ✓ @RequestBody and @ResponseBody
- ✓ @RequestHeader and @ResponseHeader

@Controller
This annotation serves as a specialization of @Component, allowing for implementation classes autodetected through classpath scanning.@Controller annotation tells the Spring IOC container to treat this class as Spring MVC Controller.
To configure/ customize MVC components, Spring MVC provides an option to handle it either through Java Config or XML. Add @EnableWebMvc will import the Spring MVC configuration from.WebMvcConfigurationSupport For XML based configuration use the
<context:component-scan base-package="com.praveen">
@Controller

```
public class SpringMVCController {
//HTTP Mappings
}
```

@RequestMapping

@RequestMapping marks request handler methods inside @Controller classes.
Consumes – The consumable media types of the mapped request, narrowing the
primary mapping. (e.g. @RequestMapping(consumes = {"application/json",
"application/xml"})).
method – The HTTP request methods to map (e.g. method =
{RequestMethod.GET,RequestMethod.POST}).
header – The headers of the mapped request.
name – the name of the mapping.
value – The primary mapping expressed by this annotation
produces – The producible media types of the mapped request.

```
@Controller
public class SpringMVCController {
@RequestMapping(value = {
"/greetings",
"/hello-world"}, method = {RequestMethod.GET,RequestMethod.POST},
consumes = {"application/json","application/xml"},
produces = { "application/json"},headers = {"application/json"
})
public String helloWorld() {
return "Hello";
}
}
```

This annotation can be used both at the class and at the method level.

@RequestParam

Annotation which shows that it binds a method parameter to a web request parameter.
Request parameters passed by the browser/client as part of the HTTP request, the
@RequestParam annotation help to map these parameters easily at the controller level.

```
@GetMapping("/fetchId")
public String fetchId(@RequestParam("id") String id) {
//}
```

With @RequestParam we can specify default value when Spring finds no or empty
value in the request.

```
public String fetchId(@RequestParam(defaultValue = "1") String id){}
```

@PathVariable

This annotation shows that a method parameter bound to a URI template variable. We
specify the variable as part of the @RequestMapping and bind a method argument with
@PathVariable. Let's take an example where we want to pass productCode as part of
the URI and not request parameter.

```
@GetMapping("/products/{id}")
public String getProduct(@PathVariable("id") String id) {
//

}
```

a) The variable name in the @PathVariable annotation is optional if the name of the part in the template matches the name of the method argument. For the above example, we can omit "id" from the @PathVariable annotation.

```
@GetMapping("/products/{id}")
public String getProduct(@PathVariable String id) {
//

}
```

b) Use "require" parameter to mark the path variable as an optional.

```
@GetMapping("/products/{id}")
public String getProduct(@PathVariable(required = false) String id) {
//

}
```

Here id is optional.

@RequestBody

The @RequestBody annotation showing a method parameter bound to the body of the web request. It passes the body of the request through an HttpMessageConverter to resolve the method argument depending on the content the request.

```
@PostMapping("/product/save") public String saveProduct(@RequestBody Product product){}
```

@ResponseBody

The @ResponseBody Annotation that shows a method return value bound to the web response body. Supported for annotated handler methods. Spring treats the result of the method as the response itself.
```
@GetMapping("/products/{id}")
public @ResponseBody Product saveProduct(@PathVariable("id") String id) {
//

}
```

@ModelAttribute

@ModelAttribute refers to a property of the Model object in Spring MVC. This ModelAttribute annotation binds a method parameter or method return value to a named model attribute, exposed to a web view.

```
@PostMapping("/customer-registration")
```

```
public String processForm(@ModelAttribute("customer") Customer customer) {}
```

The annotation is used to define objects which should be part of a Model. So if you want to have a Customer object referenced in the Model you can use the following method:

```
@ModelAttribute("customer")
public Person getCustomer() {
//
}
```

we don't have to specify the model key, Spring uses the method's name by default

```
@ModelAttribute
public Person getCustomer() {
//
}
```

Spring JDBC

Spring JDBC is designed on top of JDBC API and used to simplify the operations of JDBC.

Spring JDBC Advantages

- ✓ Connection Management has been taken care by the Spring container.

  Not required to create connection object and close connection object by programmer.

- ✓ Template based program which reduces the coding lines.

Model Bean/Model Class/Model

A class which represents a database table or UI Form to store data or to transfer data at application level. It is also known as DTO(Data Transfer Object).

JdbcTemplate org.springframework.jdbc.core package

- ✓ This is a class given from Spring API which holds an reference of DataSource(I).
- ✓ The JdbcTemplate class executes SQL queries, update statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values.
- ✓ Instances of the JdbcTemplate class are threadsafe once configured.

DataSource org.springframework.jdbc.datasource package

It represent a connection object details, it is an interface given from javax.sql package.

DriverManagerDataSource

It is an implementation class of DataSource(I). Which provides a connection object to perform the JDBC operations.

The org.springframework.jdbc.support package provides SQLException translation functionality and some utility classes

Download mysql from https://dev.mysql.com/downloads/file/?id=487685

and .NET Framework 4.5.2 installer https://www.microsoft.com/en-in/download/details.aspx?id=42642

Let's develop an application using Spring JDBC,mysql and hikari

Create Table

```
CREATE TABLE Employee(
employee_id INTEGER AUTO_INCREMENT,
employee_name varchar(20) ,
email varchar(25) ,
salary DOUBLE ,
gender varchar(20) ,
PRIMARY KEY (employee_id)
);
```

Database properties and hikari connection pool properties

```
database.properties ⊠
1 db.driver=com.mysql.cj.jdbc.Driver
2 db.url=jdbc:mysql://root:password@localhost:3306/praveendb?reconnect=true
3 db.username=root
4 db.password=password
5
6 spring.datasource.hikari.connectionTimeout=30000
7 spring.datasource.hikari.idleTimeout=600000
8 spring.datasource.hikari.maxLifetime=1800000
9 spring.datasource.hikari.maximumPoolSize=5
```

Create AppConfig class for fetching the database properties and seeting the same to datasource

```java
package com.praveen.config;

import java.beans.PropertyVetoException;

@Configuration
@ComponentScan("com.praveen")
@PropertySource("classpath:database.properties")
public class AppConfig {

    @Value("${db.driver}")
    private String DB_DRIVER;

    @Value("${db.password}")
    private String DB_PASSWORD;

    @Value("${db.url}")
    private String DB_URL;

    @Value("${db.username}")
    private String DB_USERNAME;

    @Bean
    public DataSource datasource() throws PropertyVetoException {
        HikariDataSource datasource = new HikariDataSource();
        datasource.setDriverClassName(DB_DRIVER);
        datasource.setJdbcUrl(DB_URL);
        datasource.setUsername(DB_USERNAME);
        datasource.setPassword(DB_PASSWORD);
        return datasource;
    }
}
```

Create Model class Employee

```java
package com.praveen.model;

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

@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Employee {

    private int employeeId;
    private String employeeName;
    private String email;
    private Double salary;
    private String gender;

}
```

EmployeeDAO Interface

```java
package com.praveen.dao;

import java.util.List;

import com.praveen.model.Employee;

public interface EmployeeDAO {

    public abstract void createEmployee(Employee employee);
    public abstract Employee getEmployeeById(int employeeId);
    public abstract void deleteEmployeeById(int employeeId);
    public abstract void updateEmployeeEmailById(String newEmail,int employeeId);
    public abstract List<Employee> getAllEmployeesDetails();
}
```

EmployeeRowMapper

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.

```java
package com.praveen.dao.impl;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

import com.praveen.model.Employee;

public class EmployeeRowMapper implements RowMapper<Employee> {

    @Override
    public Employee mapRow(ResultSet rs, int rowNum) throws SQLException {

        Employee employee = new Employee();
        employee.setEmail(rs.getString("email"));
        employee.setEmployeeId(rs.getInt("employee_id"));
        employee.setEmployeeName(rs.getString("employee_name"));
        employee.setSalary(rs.getDouble("salary"));
        employee.setGender(rs.getString("gender"));
        return employee;
    }
}
```

EmployeeService interface

```java
package com.praveen.service;

import java.util.List;

import com.praveen.model.Employee;

public interface EmployeeService {

    public abstract void addEmployee(Employee employee);
    public abstract Employee fetchEmployeeById(int employeeId);
    public abstract void deleteEmployeeById(int employeeId);
    public abstract void updateEmployeeEmailById(String newEmail,int employeeId);
    public abstract List<Employee> getAllEmployeesInfo();

}
```

EmployeeDAOImpl class

We will use jdbcTemplate class to execute the SQL queries, iterates over the ResultSet , and retrieves the called values, updates the instructions and procedure calls, "catches" the exceptions, and translates them into the exceptions defined in the org.springframwork.dao package**.**

```java
package com.praveen.dao.impl;

import java.util.List;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Repository;

import com.praveen.dao.EmployeeDAO;
import com.praveen.model.Employee;

@Repository
public class EmployeeDAOImpl implements EmployeeDAO {

    private JdbcTemplate jdbcTemplate;

    @Autowired
    public EmployeeDAOImpl(DataSource datasource) {
        this.jdbcTemplate = new JdbcTemplate(datasource);
    }

    @Override
    public Employee getEmployeeById(int employeeId) {
        String SQL = "SELECT * FROM EMPLOYEE WHERE employee_id=?";
        Employee employee = this.jdbcTemplate.queryForObject(SQL, new EmployeeRowMapper(), employeeId);
        return employee;
    }
```

```java
    @Override
    public void deleteEmployeeById(int employeeId) {
        String SQL = "DELETE FROM EMPLOYEE WHERE employee_Id=?";
        int update = this.jdbcTemplate.update(SQL, employeeId);
        if (update > 0)
            System.out.println("Employee is deleted..");
    }

    @Override
    public void updateEmployeeEmailById(String newEmail, int employeeId) {
        String SQL = "UPDATE EMPLOYEE set email=? WHERE employee_Id=?";
        int update = this.jdbcTemplate.update(SQL, newEmail, employeeId);
        if (update > 0)
            System.out.println("Email is updated..");

    }

    @Override
    public List<Employee> getAllEmployeesDetails() {
        String SQL = "SELECT * FROM EMPLOYEE";
        return this.jdbcTemplate.query(SQL, new EmployeeRowMapper());
    }

    @Override
    public void createEmployee(Employee employee) {
        int update = this.jdbcTemplate.update(
                "INSERT INTO EMPLOYEE(employee_name,email,gender,salary) VALUES(?,?,?,?)",
                employee.getEmployeeName(), employee.getEmail(), employee.getGender(), employee.getSalary());
        if (update > 0)
            System.out.println("Employee is created...");
    }
}
```

EmployeeServiceImpl class

```java
package com.praveen.service.impl;

import java.util.List;

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

import com.praveen.dao.EmployeeDAO;
import com.praveen.model.Employee;
import com.praveen.service.EmployeeService;

@Service("employeeService")
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeDAO employeeDAO;

    public void setEmployeeDAO(EmployeeDAO employeeDAO) {
        this.employeeDAO = employeeDAO;
    }

    @Override
    public void addEmployee(Employee employee) {
        employeeDAO.createEmployee(employee);
    }

    @Override
    public Employee fetchEmployeeById(int employeeId) {
        return employeeDAO.getEmployeeById(employeeId);
    }

    @Override
    public void deleteEmployeeById(int employeeId) {
        employeeDAO.deleteEmployeeById(employeeId);
    }
```

```java
    @Override
    public void updateEmployeeEmailById(String newEmail, int employeeId) {
        employeeDAO.updateEmployeeEmailById(newEmail, employeeId);
    }

    @Override
    public List<Employee> getAllEmployeesInfo() {
        return employeeDAO.getAllEmployeesDetails();
    }

}
```

Now let's write EmployeeTest to test the jdbc template functionality

```java
package com.praveen.test;

import java.util.List;

public class EmployeeTest {
    public static void main(String[] args) {
        try (AbstractApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);) {
            EmployeeService employeeService = ctx.getBean("employeeService", EmployeeServiceImpl.class);
            createEmployee(employeeService);
            // getEmployeeById(employeeService);
            // fetchAllEmployeesInfo(employeeService);
            // employeeService.updateEmployeeEmailById("mnp3pk1@gmail.com", 1);
            //employeeService.deleteEmployeeById(2);
        }
    }

    private static void fetchAllEmployeesInfo(EmployeeService employeeService) {
        List<Employee> empList = employeeService.getAllEmployeesInfo();
        for (Employee employee : empList) {
            System.out.println(employee.getEmployeeId() + "\t" + employee.getEmployeeName() + "\t" + employee.getEmail()
                    + "\t" + employee.getGender() + "\t" + employee.getSalary());
        }
    }

    private static void getEmployeeById(EmployeeService employeeService) {
        Employee employee = employeeService.fetchEmployeeById(1);
        System.out.println(employee.getEmployeeId() + "\t" + employee.getEmployeeName());
    }

    private static void createEmployee(EmployeeService employeeService) {
        Employee employee = Employee.builder().email("mnp3pk@gmail.com.com").employeeName("Praveen").gender("Male")
                .salary(190000.00).build();
        employeeService.addEmployee(employee);
    }
}
```

Output

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Employee is created...
```

employee_id	employee_name	email	salary	gender
1	Praveen	mnp3pk@gmail.com.com	190000	Male

Spring with ORM Frameworks

Spring provides API to easily integrate Spring with ORM frameworks such as Hibernate, JPA (Java Persistence API), JDO (Java Data Objects), Oracle Toplink and iBATIS.

Advantages of Spring ORM

There are a lot of advantage of Spring framework in respect to ORM frameworks. There are as follows:

- ✓ **Less coding is required**: By the help of Spring framework, you don't need to write extra codes before and after the actual database logic such as getting the connection, starting transaction, commiting transaction, closing connection etc.
- ✓ **Easy to test**: Spring's IOC approach makes it easy to test the application.
- ✓ **Better exception handling**: Spring framework provides its own API for exception handling with ORM framework.
- ✓ **Integrated transaction management**: By the help of Spring framework, we can wrap our mapping code with an explicit template wrapper class or AOP style method interceptor.

Spring ORM integration with Hibernate

Let's consider the same example which is used in Spring JDBC and implement the same using Spring ORM with hibernate using hibernate template,mysql & Hikari connection pool.

Please refer above Spring JDBC article for mysql setup, database.properties and table creation.

Let's create Entity class

```
package com.praveen.entity;

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

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

@Entity
@Table(name = "employee")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Employee {
    @Id
    private int employee_Id;
    private String employee_Name;
    private String email;
    private Double salary;
    private String gender;

}
```

Let's create a configuration class

```java
package com.praveen.utils;

import java.beans.PropertyVetoException;

@Configuration
@ComponentScan("com.praveen")
@EnableTransactionManagement
@PropertySource("classpath:database.properties")
public class DataSourceUtil {
    @Value("${db.driver}")
    private String DB_DRIVER;

    @Value("${db.password}")
    private String DB_PASSWORD;

    @Value("${db.url}")
    private String DB_URL;

    @Value("${db.username}")
    private String DB_USERNAME;

    @Bean
    public DataSource datasource() throws PropertyVetoException {
        HikariDataSource datasource = new HikariDataSource();
        datasource.setDriverClassName(DB_DRIVER);
        datasource.setJdbcUrl(DB_URL);
        datasource.setUsername(DB_USERNAME);
        datasource.setPassword(DB_PASSWORD);
        return datasource;
    }
}
```

Lets create a hibernate template utility class to set the session factory

```java
package com.praveen.utils;

import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.hibernate5.HibernateTemplate;

@Configuration
public class HibernateTemplateUtil {
    @Autowired
    private SessionFactory sessionFactory;

    @Bean
    public HibernateTemplate hibernateTemplate() {
        HibernateTemplate hibernateTemplate = new HibernateTemplate();
        hibernateTemplate.setSessionFactory(sessionFactory);
        return hibernateTemplate;
    }

}
```

Let's create session factory utility class

```java
package com.praveen.utils;

import java.util.Properties;

@Configuration
public class SesionFactoryUtil {
    @Autowired
    private DataSource dataSource;

    @Bean
    public LocalSessionFactoryBean sessionFactory() {
        LocalSessionFactoryBean localSessionFactoryBean = new LocalSessionFactoryBean();
        localSessionFactoryBean.setDataSource(dataSource);
        localSessionFactoryBean.setHibernateProperties(properties());
        localSessionFactoryBean.setAnnotatedClasses(Employee.class);
        return localSessionFactoryBean;

    }

    private Properties properties() {
        Properties properties = new Properties();

        properties.setProperty("hibernate.show_sql", "true");
        return properties;

    }
}
```

Let's create transaction manager utility class

```java
package com.praveen.utils;

import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.hibernate5.HibernateTransactionManager;

@Configuration
public class TransactionManagerUtil {
    @Autowired
    private SessionFactory sessionFactory;

    @Bean
    public HibernateTransactionManager transactionManager() {
        HibernateTransactionManager hibernateTransactionManager = new HibernateTransactionManager();
        hibernateTransactionManager.setSessionFactory(sessionFactory);
        return hibernateTransactionManager;

    }
}
```

Let's create EmployeeDAO class

```java
package com.praveen.dao;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.hibernate5.HibernateTemplate;

import com.praveen.entity.Employee;

@Configuration
public class EmployeeDAO {
    @Autowired
    private HibernateTemplate hibernateTemplate;

    public void save(Employee employee) {
        hibernateTemplate.save(employee);
        System.out.println("Employee is created...");
    }

    public void delete(Employee employee) {
        hibernateTemplate.delete(employee);
        System.out.println("Employee is deleted...");
    }

    public void update(Employee employee) {
        hibernateTemplate.update(employee);
        System.out.println("Employee is deleted...");
    }

    public List<Employee> getAll() {
        return hibernateTemplate.loadAll(Employee.class);
    }
}
```

Let's create the Employee Service class

```java
package com.praveen.service;

import java.util.List;

import javax.transaction.Transactional;

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

import com.praveen.dao.EmployeeDAO;
import com.praveen.entity.Employee;

@Service
public class EmployeeService {
    @Autowired
    private EmployeeDAO employeeDAO;

    @Transactional
    public void save(Employee employee) {
        employeeDAO.save(employee);
    }

    public List<Employee> getAll() {
        return employeeDAO.getAll();
    }

}
```

Let's create EmployeeTest class to test the springorm using hibernate component

```java
package com.praveen.test;

import java.util.List;

public class EmployeeTest {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext container = new AnnotationConfigApplicationContext(DataSourceUtil.class);
        EmployeeService employeeService = container.getBean(EmployeeService.class);
        createEmployee(employeeService);
        fetchAllEmployeesInfo(employeeService);
        container.close();
    }

    private static void createEmployee(EmployeeService employeeService) {
        Employee employee = new Employee();
        employee.setEmail("mnp3pk1@gmail.com.com");
        employee.setEmployee_Name("Praveen");
        employee.setGender("Male");
        employee.setSalary(190000.00);

        employeeService.save(employee);
    }

    private static void fetchAllEmployeesInfo(EmployeeService employeeService) {
        List<Employee> empList = employeeService.getAll();
        for (Employee employee : empList) {
            System.out.println(employee.getEmployee_Id() + "\t" + employee.getEmployee_Name() + "\t" + employee.getEmail()
                    + "\t" + employee.getGender() + "\t" + employee.getSalary());
        }
    }
}
```

Output

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Oct 11, 2019 12:32:15 AM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {5.2.16.Final}
Oct 11, 2019 12:32:15 AM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
Oct 11, 2019 12:32:15 AM org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
Oct 11, 2019 12:32:18 AM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQL57Dialect
Employee is created...
Hibernate: insert into employee (email, employee_Name, gender, salary, employee_Id) values (?, ?, ?, ?, ?)
Oct 11, 2019 12:32:22 AM org.hibernate.internal.SessionImpl createCriteria
WARN: HHH90000022: Hibernate's legacy org.hibernate.Criteria API is deprecated; use the JPA javax.persistence.criteria.CriteriaQuery instead
Hibernate: select this_.employee_Id as employee1_0_0_, this_.email as email2_0_0_, this_.employee_Name as employee3_0_0_, this_.gender as gender4_0_0_, this_.salary as s
1       Praveen mnp3pk1@gmail.com.com   Male    190000.0
```

employee_id	employee_name	email	salary	gender
1	Praveen	mnp3pk1@gmail.com.com	190000	Male

B l o g :   h t t p s : / / p r a v e e n n a g a t e c h . b l o g s p o t . c o m
F a c e b o o k   G r o u p :   https://www.facebook.com/groups/268426377837151
G i t h u b   r e p o :   https://github.com/praveennaga
E m a i l :   opraveennaga@gmail.com

Spring ORM Integration with EclipseLink

Let's consider the same example which is used in Spring JDBC and implement the same using Spring ORM with EclipseLink using mysql.

Please refer Spring JDBC article for mysql database setup and table creation.

Let's create Entity class

```java
package com.praveen.entity;

import javax.persistence.Entity;

@Entity
@Table(name = "employee")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Employee {
    @Id
    private int employee_Id;
    private String employee_Name;
    private String email;
    private Double salary;
    private String gender;

}
```

Let's create EntityManagerFactoryUtil class

```java
package com.praveen.utils;

import org.springframework.context.annotation.Bean;

@Configuration
@ComponentScan(basePackages="com.praveen")
@EnableTransactionManagement
public class EntityManagerFactoryUtil {

    @Bean
    public LocalEntityManagerFactoryBean entityManagerFactory() {
        LocalEntityManagerFactoryBean localEntityManagerFactoryBean = new LocalEntityManagerFactoryBean();
        localEntityManagerFactoryBean.setPersistenceUnitName("SpringJpaEclipseLinkORM");
        return localEntityManagerFactoryBean;

    }
}
```

Let's create JpaTransactionManagerUtil class

```java
package com.praveen.utils;

import javax.persistence.EntityManagerFactory;

@Configuration
public class JpaTransactionManagerUtil {
    @PersistenceUnit(name = "SpringJpaEclipseLinkORM")
    private EntityManagerFactory entityManagerFactory;

    @Bean
    public JpaTransactionManager transactionManager() {
        JpaTransactionManager jpaTransactionManager = new JpaTransactionManager();
        jpaTransactionManager.setEntityManagerFactory(entityManagerFactory);
        return jpaTransactionManager;
    }
}
```

Let's create persistence.xml

```xml
X persistence.xml ⊠
1 <persistence xmlns="http://xmlns.jcp.org/xml/ns/persistence"
2     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3     xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
4 http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd"
5     version="2.1">
6     <persistence-unit name="SpringJpaEclipseLinkORM"
7         transaction-type="RESOURCE_LOCAL">
8         <class>com.praveen.entity.Employee</class>
9         <properties>
10            <property name="javax.persistence.jdbc.url"
11                value="jdbc:mysql://root:password@localhost:3306/praveendb?reconnect=true" />
12            <property name="javax.persistence.jdbc.user" value="root" />
13            <property name="javax.persistence.jdbc.password" value="password" />
14            <property name="javax.persistence.jdbc.driver"
15                value="com.mysql.cj.jdbc.Driver" />
16        </properties>
17    </persistence-unit>
18 </persistence>
```

Let's create EmployeeDAO class

```java
package com.praveen.dao;

import javax.persistence.EntityManager;

@Repository
public class EmployeeDAO {
    @PersistenceContext
    private EntityManager entityManager;

    public void save(Employee employee) {
        entityManager.persist(employee);
        System.out.println("Employee is created...");
    }

    public void delete(Employee employee) {
        entityManager.remove(employee);
        System.out.println("Employee is deleted...");
    }

    public void update(Employee employee) {
        entityManager.merge(employee);
        System.out.println("Employee is deleted...");
    }

    public Employee getEmployee(int empId) {
        return entityManager.find(Employee.class,empId);
    }
}
```

Let's create EmployeeService class

```java
package com.praveen.service;

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

@Service("employeeService")
public class EmployeeService {
    @Autowired
    private EmployeeDAO employeeDAO;

    @Transactional
    public void save(Employee employee) {
        employeeDAO.save(employee);
    }

    public Employee getEmployee(int empId) {
        return employeeDAO.getEmployee(empId);
    }

}
```

Let's create EmployeeTest class

```java
package com.praveen.test;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class EmployeeTest {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext container = new AnnotationConfigApplicationContext(
                EntityManagerFactoryUtil.class);
        EmployeeService employeeService = container.getBean("employeeService",EmployeeService.class);
        createEmployee(employeeService);
        getEmployee(employeeService);
        container.close();
    }

    private static void createEmployee(EmployeeService employeeService) {
        Employee employee = new Employee();
        employee.setEmail("mnp3pk1@gmail.com.com");
        employee.setEmployee_Name("Praveen");
        employee.setGender("Male");
        employee.setSalary(190000.00);

        employeeService.save(employee);
    }

    private static void getEmployee(EmployeeService employeeService) {
        Employee employee = employeeService.getEmployee(0);
        System.out.println(employee.getEmployee_Id() + "\t" + employee.getEmployee_Name() + "\t" + employee.getEmail()
                + "\t" + employee.getGender() + "\t" + employee.getSalary());
    }
}
```

Output

```
Oct 11, 2019 1:03:26 AM org.springframework.orm.jpa.AbstractEntityManagerFactoryBean buildNativeEntityManagerFactory
INFO: Initialized JPA EntityManagerFactory for persistence unit 'SpringJpaEclipseLinkORM'
[EL Info]: 2019-10-11 01:03:27.518--ServerSession(1052253947)--EclipseLink, version: Eclipse Persistence Services - 2.5.0.v20130507-3faac2b
[EL Info]: connection: 2019-10-11 01:03:28.113--ServerSession(1052253947)--file:/D:/Praveen/workspaces/springworkspace/praveenoruganti-springorm-eclipselink-master/target
Employee is created...
0       Praveen mnp3pk1@gmail.com.com   190000.0
Oct 11, 2019 1:03:28 AM org.springframework.orm.jpa.AbstractEntityManagerFactoryBean destroy
INFO: Closing JPA EntityManagerFactory for persistence unit 'SpringJpaEclipseLinkORM'
[EL Info]: connection: 2019-10-11 01:03:28.331--ServerSession(1052253947)--file:/D:/Praveen/workspaces/springworkspace/praveenoruganti-springorm-eclipselink-master/target
```

employee_id	employee_name	email	salary	gender
1	Praveen	mnp3pk1@gmail.com.com	190000	Male

Spring Annotations

Annotation injection is performed before XML injection. Thus XML configuration will override that.

Core Spring framework Annotations

a) **@Required**

This annotation is applied on bean setter methods. Consider if you need to enforce a required property at configuration time if it is not set BeanInitializationException is thrown.

b) **@Autowired**

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

c) **@Qualifier**

This annotation is used along with @Autowired when you need more control of the dependency injection process.

This annotation is used to avoid confusion which occurs when you create more than one bean of the same type and want to wire only one of them with a property.

```
For example,
public interface Vehicle {
public void start();
public void stop();
}
@Component(value="car") or can be denoted as @Component @Qualifier("car")
public class Car implements Vehicle {
@Override
public void start(){
System.out.println("Car Started");
}
@Override
public void stop(){
System.out.println("Car Stopped");
}
}
@Component(value="bike")
public class Bike implements Vehicle {
@Override
public void start(){
System.out.println("Bike Started");
}
@Override
public void stop(){
System.out.println("Bike Stopped");
}
}
```

```
@Component
public class VehicleService {
@Autowired
@Qualifier("bike")
private Vehicle vehicle;
public void service(){
vehicle.start();
vehicle.stop();
}
}
```

d) **@Primary**

Indicates that a bean should be given preference when multiple candidates are qualified to autowire a single-valued dependency. If exactly one 'primary' bean exists among the candidates, it will be the autowired value.

May be used on any class directly or indirectly annotated with @Component or on methods annotated with @Bean

e) **@Configuration**
This annotation is analog for XML configuration file.

```
@Configuration
public class DataConfig {
@Bean
public Datasource source(){
Datasource source= new OracleDataSource();
source.setURL();
source.setUser();
return source;
}
}
```

f) **@ComponentScan**
This is used in conjuction with @Configuration annotation to allow spring to know the packages to scan.

```
@Configuration
@ComponentScan("com.praveen")
public class DataConfig{ }
```

g) **@Bean**
This annotation is used at method level. @Bean works with @Configuration to create spring beans.

h) **@Lazy**
When we put @Lazy annotation over the @Configuration class, it indicates that all the methods with @Bean annotation should be loaded lazily.

```
@Lazy
@Configuration
```

```
@ComponentScan(basePackages = "com.baeldung.lazy")
public class AppConfig {

@Bean
public Region getRegion(){
return new Region();
}

@Bean
public Country getCountry(){
return new Country();
}

}
```

With @Autowired, in order to initialize a lazy bean, we reference it from another one.

```
public class Region {

@Lazy
@Autowired
private City city;

public Region() {
System.out.println("Region bean initialized");
}
public City getCityInstance() {
return city;
}

}
```

i) **@Value**

This annotation can be used for injecting values into fields in Spring-managed beans and it can be applied at the field or constructor/method parameter level.

Generally this is used to fetch the values from property file.

```
@Configuration
@ComponentScan("com.praveen")
@PropertySource("classpath:database.properties")
public class AppConfig {

@Value("${db.driver}")
private String DB_DRIVER;

@Value("${db.password}")
private String DB_PASSWORD;

@Value("${db.url}")
private String DB_URL;

@Value("${db.username}")
private String DB_USERNAME;
```

```
@Bean
public DataSource datasource() throws PropertyVetoException {

HikariDataSource datasource = new HikariDataSource();
datasource.setDriverClassName(DB_DRIVER);
datasource.setJdbcUrl(DB_URL);
datasource.setUsername(DB_USERNAME);
datasource.setPassword(DB_PASSWORD);
return datasource;
}

}
```

Difference between @Autowired with @Qualifier and @Primary

If a bean has @Autowired without any @Qualifier, and multiple beans of the type exist, the candidate bean marked @Primary will be chosen, i.e. it is the default selection when no other information is available, i.e. when @Qualifier is missing.

Difference between @Autowired and @Resource

The main difference is that @Autowired wires per type and @Resource wires per bean name. But @Autowired in combination with @Qualifier also autowires by name.

@Resource is related to Java specification Request(JSR 250) and is a standard annotation can be used in different frameworks and technologies i..e @Resource(name="praveen")
If i don't provide name i.e.. @Resource, by default fetches the bean based on member variable name.

There are 2 beans related to JSR specific to bean initialize and destroy i.e.. @PostConstruct and @PreDestroy are useful for init and destory of beans.

Difference between @Component and @Configuration with @Bean

@Bean is used to explicitly declare a single bean rather than letting spring do it automatically like we did with @Component.It decouples the declaration of the bean from the class definition and lets you create and configure beans exactly how you choose.

@Bean can't be used at class level rather it is placed at method level in @Configuration class.

Spring Framework stereo type annotations

a) @Component
This annotation is used on classes to indicate a spring component @Component annotation marks the java class as a bean or say component so that the component scanning mechanism of spring can add into the application context.

There is an implicit one to one mapping between the annotation class and bean(i.e. one bean per class). Control of wiring is limited with this approach since it is purely declarative.

b) **@Controller**

This annotation is used on class to indicate the class is a spring controller.

c) **@Service**

This annotation is used on class to indicate the class as a service where you can mention business logic perform calculations and call external APIs.

d) **@Repository**

This is used on java class which directly access the database.

Spring Boot annotations

a) **@EnableAutoConfiguration**

This is usually placed in main application class. This implicitly defines a base search package.This annotation tells spring boot to start adding beans based on classpath settings, other beans and various property settings.

b) **@SpringBootApplication**

This annotation is used on the application class while setting up a springboot project. This class needs to be present in base package.

This annotation adds all the following.

- ✓ @Configuration
- ✓ @EnableAutoConfiguration
- ✓ @ComponentScan

Spring MVC and REST annotations

a) **@Controller** acts as a controller class.

b) **@RequestMapping**

This annotation is used both at class and method level. This is used to map web request on to the specific handler classes and handler methods.

When @RequestMapping is used on class level it creates a base URI for which controller will be used.

When this annotation is used on methods it will give you the URI on which the handler methods will be executed.

@Controller
@RequestMapping("/welcome")
public class WelcomeController{
@RequestMapping(method=RequestMethod.GET)
public string welcomeAll(){
return "welcome all";
}
}
This annotation can be used in spring MVC as well.

@RequestMapping with Mutliple URIs

```
@RestController
@RequestMapping("/home")
public class IndexController{
@RequestMapping(value={"/page","page*","view/*","*/msg"})
String indexMultipleMapping(){
return "Hello from index multiple mapping";
}
}
```

Composed @RequestMapping variants

```
@GetMapping
@PostMapping
@PutMapping
@PatchMapping
@DeleteMapping
```

@RequestMapping with @RequestParam

The @RequestParam is used with @RequestMapping to bind a web request parameter to the parameter of handler method.

@RequestParam can be used with or without value

```
@RestController
@RequestMapping("/home")
public class IndexController {
@RequestMapping(value="/id")
String getIdByValue(@RequestParam("id") String personId){
System.out.println("Id is "+ personId);
return "Get ID from query string or URL with value element";
}
@RequestMapping(value="/personId")
String getId(@RequestParam String personId){
System.out.println("Id is "+ personId);
return "Get ID from query string of URL without value element";
}
}
```

The value element of @RequestParam can be omitted if the request param and handler method parameter names are same.

produces and consumes

```
@RestController
@RequestMapping("/home")
public class IndexController {
@RequestMapping(value="/prod", produces={"application/json")
@ResponseBody
String getProduces(){
```

```
return "produces attribute";
}
@RequestMapping(value="/cons", consumes={"application/json","application/xml")
String getConsumes(){
return "consumes attribute";
}
}
```

@RequestMapping with @PathVariable to handle dynamic URI's

```
@RestController
@RequestMapping("/home")
public class IndexController{
@RequestMapping(value="/fetch/{id}",method=RequestMethod.GET)
String getDynamicUriValue(@PathVariable String id){
System.out.println("ID is "+id);
return "Dynamic URI parameter fetched";
}
@RequestMapping(value = "/fetch/{id:[\\d]+}", method = RequestMethod.GET)
String getDynamicUriValueRegex(@PathVariable String id){
System.out.println("ID is "+id);
return "Dynamic URI parameter fetched with regex";
```

Spring Data JPA Using Hibernate

What is JPA

The Java Persistence API provides a specification for persisting, reading, and managing data from your Java object to relational tables in the database.

The problem with JPA Using Hibernate which we learnt in our earlier article is there are lot of duplication between Repositories.

The other problem is proliferation of data stores i.e.. many are coming up like SQL, NO SQL etc.

Spring Data provises the abstraction support to handle the above problems.

What is Spring Data JPA

Provides repository support for the Java Persistence API(JPA).

The goal of Spring Data repository abstraction is to significantly reduce the amount of boilerplate code required to implement data access layers for various persistence
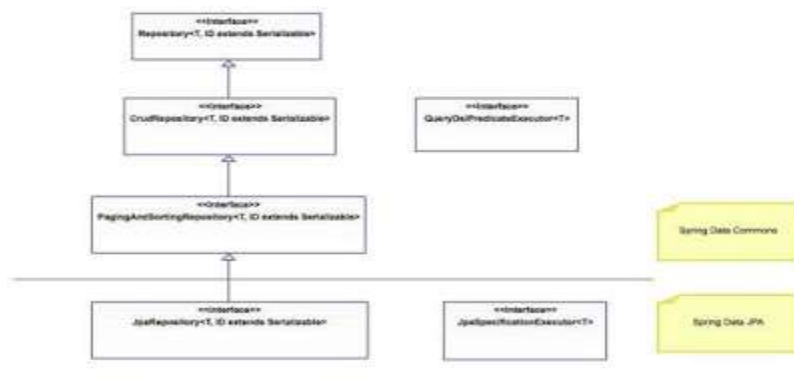
stores.

Spring Data JPA is not a JPA provider. It is a library/framework that adds an extra layer of abstraction on the top of our JPA provider (like Hibernate).

Hibernate is a JPA implementation, while Spring Data JPA is a JPA Data Access Abstraction.

Lets see how Hibernate,EclipseLink and Spring Data JPA categorizes using below diagram.

Core API

Let's develop an application having springdatajpa using hibernate

Let's create an Entity class

```java
package com.praveen.entity;

import javax.persistence.Entity;

@Entity
@Table(name = "employee")
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Employee {
    @Id
    private int employee_Id;
    private String employee_Name;
    private String email;
    private Double salary;
    private String gender;
}
```

database.properties

```
database.properties ⊠
 1 db.driver=com.mysql.cj.jdbc.Driver
 2 db.url=jdbc:mysql://root:password@localhost:3306/praveendb?reconnect=true
 3 db.username=root
 4 db.password=password
 5
 6 spring.datasource.hikari.connectionTimeout=30000
 7 spring.datasource.hikari.idleTimeout=600000
 8 spring.datasource.hikari.maxLifetime=1800000
 9 spring.datasource.hikari.maximumPoolSize=5
10 hibernate.dialect=org.hibernate.dialect.MySQLDialect
11 hibernate.show_sql=true
12 hibernate.hbm2ddl.auto=update
13 hibernate.cache.use_second_level_cache=false
14 hibernate.cache.use_query_cache=false
```

Let's create PersistenceJPAConfig class

```java
package com.praveen.config;

import java.util.Properties;

@Configuration
@EnableTransactionManagement
@PropertySource({
    "classpath:database.properties"
})
@ComponentScan({
    "com.praveen"
})
@EnableJpaRepositories(basePackages = "com.praveen")
public class PersistenceJPAConfig {

    @Autowired
    private Environment env;

    public PersistenceJPAConfig() {
        super();
    }

    @Bean
    public LocalContainerEntityManagerFactoryBean entityManagerFactory() {
        final LocalContainerEntityManagerFactoryBean entityManagerFactoryBean = new LocalContainerEntityManagerFactoryBean();
        entityManagerFactoryBean.setDataSource(dataSource());
        entityManagerFactoryBean.setPackagesToScan(new String[] {
            "com.praveen"
        });

        final HibernateJpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
        entityManagerFactoryBean.setJpaVendorAdapter(vendorAdapter);
        entityManagerFactoryBean.setJpaProperties(additionalProperties());

        return entityManagerFactoryBean;

    }

    final Properties additionalProperties() {
        final Properties hibernateProperties = new Properties();
        hibernateProperties.setProperty("hibernate.hbm2ddl.auto", env.getProperty("hibernate.hbm2ddl.auto"));
        hibernateProperties.setProperty("hibernate.dialect", env.getProperty("hibernate.dialect"));
        hibernateProperties.setProperty("hibernate.cache.use_second_level_cache", env.getProperty("hibernate.cache.use_second_level_cache"));
        hibernateProperties.setProperty("hibernate.cache.use_query_cache", env.getProperty("hibernate.cache.use_query_cache"));
        // hibernateProperties.setProperty("hibernate.globally_quoted_identifiers", "true");
        return hibernateProperties;
    }

    @Bean
    public DataSource dataSource() {
        final HikariDataSource datasource = new HikariDataSource();
        datasource.setDriverClassName(env.getProperty("db.driver"));
        datasource.setJdbcUrl(env.getProperty("db.url"));
        datasource.setUsername(env.getProperty("db.username"));
        datasource.setPassword(env.getProperty("db.password"));
        return datasource;
    }
    @Bean
    public PlatformTransactionManager transactionManager(final EntityManagerFactory emf) {
        final JpaTransactionManager transactionManager = new JpaTransactionManager();
        transactionManager.setEntityManagerFactory(emf);
        return transactionManager;
    }

    @Bean
    public PersistenceExceptionTranslationPostProcessor exceptionTranslation() {
        return new PersistenceExceptionTranslationPostProcessor();
    }

}
```

Let's create Employee Repository class

```java
package com.praveen.repository;

import org.springframework.data.jpa.repository.JpaRepository;

import com.praveen.entity.Employee;

public interface EmployeeRepository extends JpaRepository < Employee, Long > {

}
```

Let's create EmployeeService class

```java
package com.praveen.service;

import java.util.List;

@Service
public class EmployeeService {
    @Autowired
    private EmployeeRepository employeeRepository;
    @Transactional
    public void save(Employee employee) {
        employeeRepository.save(employee);
        System.out.println("Employee is created...");
    }

    public List<Employee> getAll(){
        return employeeRepository.findAll();
    }

}
}
```

Let's create EmployeeTest class

```java
package com.praveen.test;

import java.util.List;

public class EmployeeTest {
    public static void main(String[] args) {
        AnnotationConfigApplicationContext container = new AnnotationConfigApplicationContext(PersistenceJPAConfig.class);
        EmployeeService employeeService = container.getBean(EmployeeService.class);
        createEmployee(employeeService);
        fetchAllEmployeesInfo(employeeService);
        container.close();
    }

    private static void createEmployee(EmployeeService employeeService) {
        Employee employee = new Employee();
        employee.setEmail("mnp3pk1@gmail.com.com");
        employee.setEmployee_Name("Praveen");
        employee.setGender("Male");
        employee.setSalary(190000.00);

        employeeService.save(employee);
    }

    private static void fetchAllEmployeesInfo(EmployeeService employeeService) {
        List<Employee> empList = employeeService.getAll();
        for (Employee employee : empList) {
            System.out.println(employee.getEmployee_Id() + "\t" + employee.getEmployee_Name() + "\t" + employee.getEmail()
                    + "\t" + employee.getGender() + "\t" + employee.getSalary());
        }
    }
}
```

Output

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
Oct 11, 2019 1:37:28 AM org.hibernate.jpa.internal.util.LogHelper logPersistenceUnitInformation
INFO: HHH000204: Processing PersistenceUnitInfo [
    name: default
    ...]
Oct 11, 2019 1:37:29 AM org.hibernate.Version logVersion
INFO: HHH000412: Hibernate Core {5.2.16.Final}
Oct 11, 2019 1:37:29 AM org.hibernate.cfg.Environment <clinit>
INFO: HHH000206: hibernate.properties not found
Oct 11, 2019 1:37:29 AM org.hibernate.annotations.common.reflection.java.JavaReflectionManager <clinit>
INFO: HCANN000001: Hibernate Commons Annotations {5.0.1.Final}
Oct 11, 2019 1:37:29 AM org.hibernate.dialect.Dialect <init>
INFO: HHH000400: Using dialect: org.hibernate.dialect.MySQLDialect
Employee is created...
Oct 11, 2019 1:37:32 AM org.hibernate.hql.internal.QueryTranslatorFactoryInitiator initiateService
INFO: HHH000397: Using ASTQueryTranslatorFactory
1       Praveen mnp3pk1@gmail.com.com   Male    190000.0
```

employee_id	employee_name	email	salary	gender
1	Praveen	mnp3pk1@gmail.com.com	190000	Male

You can refer code in https://github.com/praveennaga/praveennaga-spring-master

Please check out my other ebooks in https://github.com/praveennaga/PraveenNaga-Tech-Ebooks

Buy me a Cup of Coffee:
https://docs.google.com/forms/d/e/1FAIpQLSfZxtgAxzMv83uwdEXezVFfLNSKlQgu7eD
cMOeGtS2cRnOBrA/viewform?vc=0&c=0&w=1